International Journal of Research (IJR) Vol-1, Issue-9 October 2014 **ISSN 2348-6848**

Compiler

Nonika Sharma & Priyanka Sahni

Information Technology, Dronacharya College of Engg, Gurgaon, Haryana, India

Nonikasharma1@gmail.com Priyanka.sahnni@yahoo.com

ABSTRACT:

A compiler is a set of programs that transforms source code written in the source language into target language that often have a binary form called as object code. Compiler is a program that translate source code from high level programming language to low level programming language. Compiler are the kind of translators. The reason behind the transformation of source code is to create an executable program. In our research paper we are going to study about what is a compiler, its working, construction, properties, phases and its types.

KEYWORDS: Compiler, Executable Program, Source Code, Object Code, Translators, Interpreter, Transformation, Compiler Technique

INTRODUCTION

Compilation is a process that translates a program in source language into an equivalent program in the object or target language. Compiler is the detection and reporting of errors. Compilation is a fundamental concept in the production of software and link between the world of application development and the low-level world of application that is to be executed on machines. CPU or operating system in which compiled program can run on a computer is different from the one on which the compiler runs, and this type of compiler is known as a cross-compiler. A program that translates between high-level languages is called a source-to-source compiler . A language rewriter is a program that translates the form of expressions without a change of language. Compiler-compiler is the term used to refer to a parser, is a tool used to help to create the lexer and parser.

A compiler perform many of the following operations:

- Lexical
- preprocessing
- parsing
- semantic analysis (Syntax-directed translation)
- code generation
- code optimization

Faults in the program caused by incorrect compiler behavior which can be very difficult to track down and work around and the compiler implementers invest significant effort to ensure compiler correctness .An assembler is also a type of translator. It is basically a language converter as we see in the following diagram :

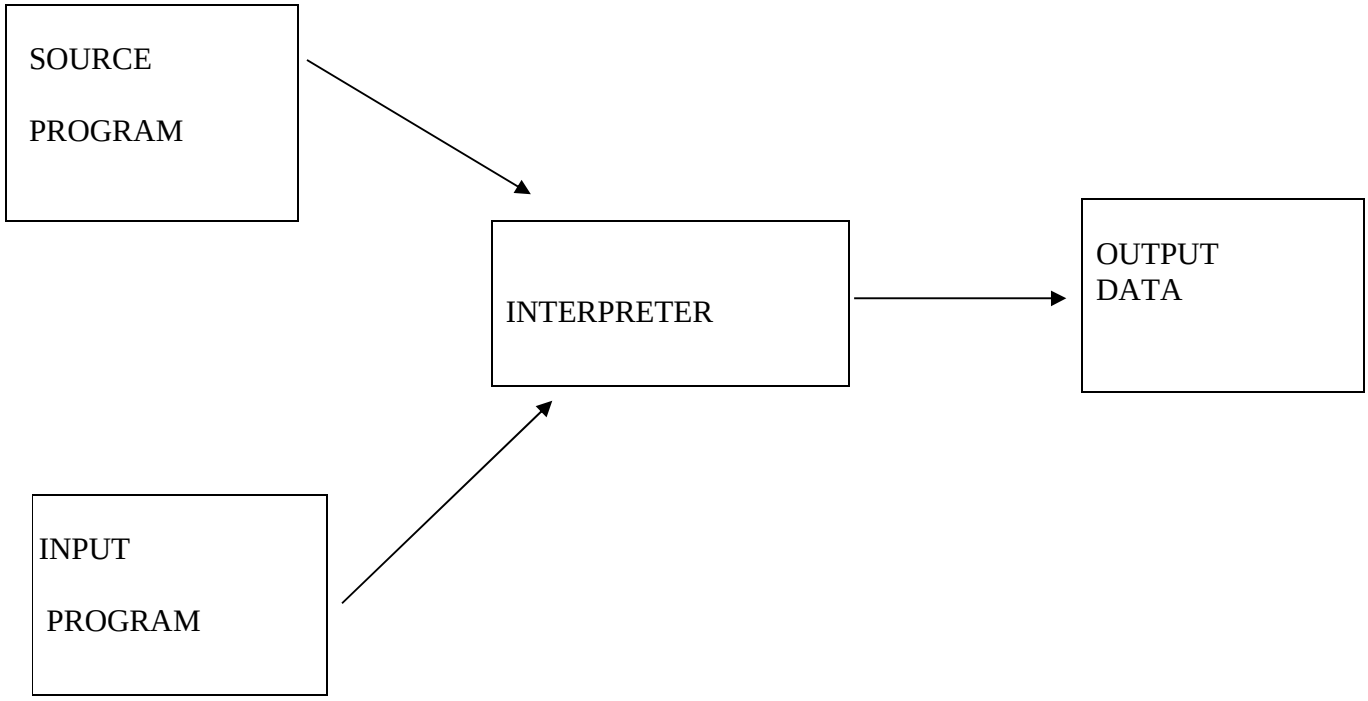
Assembly Program → Assembler → Machine Program

Types of Translator :

- Assembly Program
- Assembler
- Machine Program

INTERPRETER

An **interpreter** is same as that of a compiler, but takes both source program and input data.
Translation and execution phases of the source program are one and the same.
An interpreter can itself be interpreted, directly executed program that performs instructions written in programming language.



Many other types of languages including:

• Command-line interface languages

• Typesetting languages

• Natural languages

• Hardware description languages

• Page description languages

• Set-up or parameter files

COMPILER IN EDUCATION SYSTEM

Compiler construction and compiler optimized are the courses usually supplemented with the implementation of a

compiler for an educational programming language.

A well-known example is Niklaus Wirth's PL/0 compiler, which Wirth used to teach compiler construction in the 1970s inspite of its simplicity, the compiler introduced several influential concepts to the field:

1. Program development by stepwise refinement
2. Use of a recursive descent parser and use of EBNF to specify the syntax of a language
3. Code generator produces portable P-code
4. Use of T-diagrams in the formal description

Difference between Compiled Vs. Interpreted languages

Higher-level programming languages is a type of translation either designed as compiled language or interpreted language. However, in practice there is rarely anything about a language which requires it to be exclusively compiled or exclusively interpreted. It is the most popular or widespread implementation of a language — for instance, BASIC is sometimes called an interpreted language and C a compiled one, despite the existence of BASIC compilers and C interpreters.

Interpretation does not replace compilation completely. It only hides it from the user. Some language specifications spell out that implementations must include a compilation facility. However, there is nothing inherent in the definition of Common Lisp.

HISTORY

(EARLY DEVELOPMENT OF COMPILERS)

In Early Computers software was written in programming language. On designing of the first compiler limited memory capacity of early computers that may led to substantial technical challenges.
In 1940s. Early stored-program computers were programmed in machine language. Later, assembly languages were developed where machine instructions and memory locations were given symbolic forms.

Towards the end of the 1950s, machine-independent programming languages were first proposed and was written by Grace Hopper, in 1952, for A-0 programming language .It is more as a loader or linker than the modern notion of a compiler.

In 1952, first auto code and its compiler were developed by Alick and is considered by some to be the first compiled programming language. John Backus led FORTRAN team at IBM is credited as having introduced the first complete compiler in 1957.

In 1960, COBOL was an early language to be compiled on multiple architectures. Because of the expanding functionality supported by newer programming languages and the increasing complexity of

computer architectures, compilers become more complex.

The first self-hosting compiler capable of compiling its own source code in a high-level language and it was created in 1962 for Lisp by Tim Hart and Mike Levin at MIT.

Since 1970s it has become common practice to implement a compiler in the language it compiles, although both Pascal and C have been very popular choices for implementation language.

To build a self-hosting compiler is a bootstrapping problem—the first such compiler for a language must be compiled either by hand or a compiler written in a different language, compiled by running the compiler in an interpreter.

COMPILATION

Translation of source code into object code in computer programming language by a compiler. Compilers enabled the development of programs that are machine-independent. In the 1950s, Machine-dependent assembly language was the first higher-level language, widely used before the development of FORTRAN .

With the arrival of high-level programming languages that followed FORTRAN, such as COBOL, C, and BASIC, programmers can write machine-independent source programs. Compiler translates the high-level source programs into target programs in machine languages for the specific hardware. When target program is generated the user can execute the program.

A native compiler is one which output is planned to directly run on the same type of computer and operating system that the compiler itself runs on. The output of a cross compiler is designed to run on a different platform. Cross compilers are used when developing software for embedded systems that are not planned to support a software development environment.

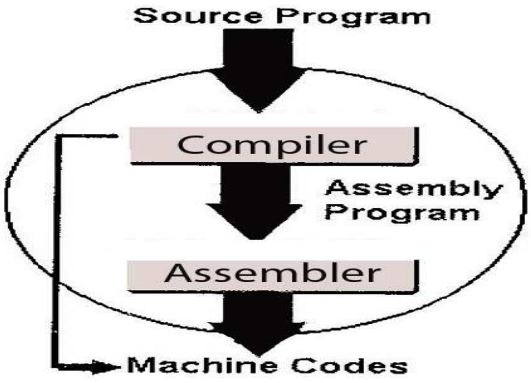
STRUCTURE OF COMPILER

Compilers link source programs in high-level languages with the underlying hardware. It verifies code syntax, generates efficient object code, performs run-time organization, and formats the output according to assembler and linker conventions.

A compiler consists of:

- The front end: It verifies syntax and semantics, and generates an intermediate representation or IR of the source code for processing by the middle-end. It performs type checking by collecting type information. It generates errors and warning, if any, in a useful way. Many aspects of the front end may include lexical analysis, syntax analysis, and semantic analysis.

- The middle end: It performs optimizations, including removal of useless or unreachable code, discovery and propagation of constant values, relocation of computation to a less frequently executed place or specialization of computation based on the context. Atlast generates another IR for the backend.
- The back end: It generates the assembly code, performing register allocation in process. Optimizes target code utilization of the hardware by figuring out how to keep parallel execution units busy, filling delay slots. Although most algorithms for optimization are in NP, heuristic techniques are well-developed.



COMPILER CONSTRUCTION

C**o**mpiler is a system program used to translate source code into machine codes. Compiler is also called as language translator. Every programming language has its own compilers or interpreters. The designing of interpreter is not more difficult like compiler. Every year many softwares are launched in market, but compiler is launched very hardly per year. It is a time consuming process. There are many several of compilers available in market such as Pascal, FORTRAN, COBOL, C, C++, C#, Java and many other high level programming and their compilers are popular among software professionals. The source program is fed into compiler. Generally compiler converts the source code into machine codes, but some compiler converts source codes into assembly language and assembler converts them into machine language. In the early days, approach taken to compiler design used to be directly exaggerated by the complexity of the processing, designing it, and the resources available. A compiler is a relatively simple language written by one person might be a single massive piece of software. When the source language is large and complex, and high quality output is required, the design may be split into a number of relatively independent phases. The division of compilation may include front , middle and back end.

The point at which these two ends meet is open to debate.

The front end is generally considered to be where syntactic and semantic processing takes place, along with translation to a lower level of representation.

The middle end is designed to perform optimizations on a form other than the source code or machine code. This machine code independence is intended to enable common optimizations to be shared between

versions of the compiler supporting different languages and target processors.

The back end takes the output from the middle. It may perform more analysis, transformations and optimizations that are for a particular computer. Then, it generates code for a particular processor and OS.

This front-end/middle/back-end approach makes it possible to combine front ends for different languages with back ends for different CPUs. Some of the practical examples of this approach are the GNU Compiler Collection, LLVM, and the Amsterdam Compiler Kit, which have multiple front-ends, shared analysis and multiple back-ends.

FRONT END

In some cases additional phases are used, notably line
reconstruction and preprocessing, but these are very rare. A detailed list of possible phases includes:

1. Line reconstruction: Languages which strop their keywords or allow arbitrary spaces within identifiers require a phase before parsing, which converts the input character

sequence to a canonical form ready for the parser.

2. Preprocessing: Some languages such as C require a preprocessing phase which supports macro substitution and conditional compilation. Typically preprocessing phase occurs before syntactic or semantic analysis.

BACK END

The back end is sometimes baffled with code generator because of the overlapped functionality of generating assembly code. Some literature uses middle end to distinguish the generic analysis and optimization phases in the back end from the machine-dependent code generators.

PHASES

Compilation is divide into six phases, each of which interacts with a symbol table manager and an error handler is called the analysis model of compilation.
There are many variants but the essential elements are the same.The compilers has some special types of procedures and these procedures are completed in pre-defined phases. When one phase is completed next phase is started. It goes on in the same process.

These phases are:

- Lexical Analyzer
- Syntax Analyzer
- Semantic Analyzer
- Intermediate code generator
- Code Generator
- Symbol table manager
- Error Handler

1. Lexical Analysis

The lexical analysis is a process to take source program and procedure lexical tokens or tokens. It is efficiently handled by lexer or lexical analyzer. Lexical analysis breaks the source code text into small pieces called tokens. Each token is a single atomic unit of the language. This phase is called scanning and the software doing lexical analysis is called a lexical analyzer or scanner..

Example:

Amount:= salary + rent;

Tokens are:

(a) Amount: Identifier

(b):=: Assignment symbol

(c) Salary: Identifier

(d) +: Operator

(e) rent: Identifier

The token is a lowest level sequence of sub-string which contains numerical constants, literal strings, operator symbols, punctuation symbols and control structures such as assignment, conditions and looping.

2.Syntax Analysis

Grouping of tokens into grammatical phrases is called syntax analysis. It involves parsing the token sequence to identify the syntactic structure of the program. This phase builds a parse tree,

which replaces the linear sequence of tokens with a tree structure built. The parse tree is frequently analyzed, augmented, and transformed by later phases in the compiler. A syntax analyzer takes tokens as input and output error message when program syntax is wrong. There are many algorithms for parsing. The most popular types of parsing are top-down parsing and bottom-up parsing.

3.Semantic Analysis

It is a process of semantic error detection in source program. Grammatically correct statements are not semantically correct that's why compiler is equipped with semantic error checking facilities. It is the stage in which the compiler adds semantic information to the parse tree and builds the symbol table and it performs semantic checks such as type checking and object binding .

4.Intermediate code generator

The intermediate code generator transforms parse tree into an intermediate language which represents source code program.

Three-Address Code is a popular type of intermediate languages.

Example:

Amount:= salary op rent

amount, salary and rent are operand and op is a binary operator.

5. Code Optimizer

It improves the output of intermediate code generator. It optimizes intermediate codes and produce fast running machines codes. There are two common optimizations: Local optimization and Loop optimization.

6. Code Generator

It is a final phases in which re-locatable machine codes or assembly codes are produced. The statements "amount: =salary + rent;" can be converted into assembly language.

The assembly language version of statement:

LOAD salary
ADD rent **STORE** amount

Assembler converts assembly codes into machine codes for CPU because CPU understands only machine codes, not any high level programming languages.

7. Symbol-table Management

The table of identifiers and their attributes are called symbol table.

8. Error handling

Each phase of compilation contains some errors. These errors are collected and noticed at the time of compilation.

COMPILER TECHNIQUES

Assembly language is a type of low-level language and a program that compiles it is more commonly known as an assembler, and the inverse program known as a disassemble. A program that translates from a low level language to a higher level know as a decompile. A program that translates high-level languages is called a

language translator, source to source translator, language converter or rewriter. Cross compiler is a program that translates into an object code format that is not supported on the compilation machine.

COMPILER ADVANTAGES

- Fast execution
- Optimize

COMPILER DISADVANTAGES

- Editing and developing of code is slower than interpreters
- It always require a complier
- Once compiled it can run on specific platform

REFERENCES

[1] Compiler textbook references A collection of references to mainstream Compiler Construction Textbooks

[2] Aho, Alfred V.; Sethi, Ravi; Ullman , Jeffrey D. (1986).Compilers: Principles, Techniques, and Tools (1st ed.).Addison-Wesley.

[3] Allen, Frances E. (September 1981). "A History of Language Processor Technology in IBM". IBM Journal of Research and Development (IBM)

[4] Allen, Randy; Kennedy, Ken (2001). Optimizing Compilers for Modern Architectures. Morgan Kaufmann Publishers ISBN .

[5] Appel, Andrew Wilson (2002). Modern Compiler Implementation in Java (2nd ed.). Cambridge University Press. ISBN.

[6] Appel, Andrew Wilson (1998). Modern Compiler Implementation in ML. Cambridge University Press.ISBN 0-521-58274-1.

[7] Bornat, Richard (1979). Understanding and Writing Compilers: A Do It Yourself Guide. Macmillan Publishing. ISBN 0-333-21732-2.

[8] Cooper, Keith D.; Torczon , Linda (2004). Engineering a Compiler. Morgan Kaufmann. ISBN 1-55860-699-8.

[9] Leverett, Bruce W; Cattell, R. G. G.; Newcomer, Joseph M.; Hobbs, S.O.; Reiner, A.H.; Schatz, B.R.; Wulf, W.A. (August 1980). "An Overview of the Production – Quality Compiler – Compiler Project.

[10] McKeeman , William Marshall; Horning, James J.; Wortman, David B. (1970). A Compiler Generator.Englewood Cliffs, NJ: Prentice-Hall.