Journal Of Harmonized Research (JOHR)

Journal Of Harmonized Research in Engineering
1(2), 2013, 91-94

ISSN 2347 – 7393

Review Article

# COMPILER AND ITS PHASES

**Shweta Garg, Shrishti Vashist, Shruti Aggarwal**

CSE Department, Dronacharya College Of Engineering,
Gurgaon, India

**Abstract**:
A compiler's job is to Lower the abstraction level, eliminate overhead from language abstractions, map source program onto hardware efficiently hide hardware weaknesses, utilize hardware strengths equal the efficiency of a good assembly programmer. The process of compiling a set of source files into a corresponding set of class files is not a simple one, but can be generally divided into three stages. Different parts of source files may proceed through the process at different rates, on an "as needed" basis.

**Keywords**: Specialization- pursuing a particular line of work, Fidelity- strict observance of promises, Immutable- unchangeable, Trivia- inconsequential, Computation- an act or process.

**Abbreviation:**-IDE- integrated development environment, IR- intermediate representation

## 1. Introduction

A compiler is a computer program that transforms source code written in a programming language into another computer language as *object code*. The name "compiler" is primarily used for programs that translate source code from a high-level programming language to a lower level language or machine code. If the compiled program can run on a computer whose CPU or operating system is different from the one on which the compiler runs, the compiler is known as a cross-compiler.

A compiler is a special program that processes statements written in a particular programming language and turns them into machine language or "code" that a computer's processor uses.

When executing , the compiler first parses all of the language statements syntactically one after the other and then, in one or more successive stages or "passes", builds the output code, making sure that statements that refer to other statements are referred to correctly in the final code. Traditionally, the output of the compilation has been

**For Correspondence:**
shivigarg101ATgmail.com
Received on: October 2013.
Accepted after revision: November 2013.
Downloaded from: www.johronline.com

called *object code* or sometimes an *object module*.

A compiler is likely to perform many or all of the following operations: lexical analysis, preprocessing, parsing,semantic analysis (Syntax-directed translation), code generation, and code optimization.

**2. Structure of a compiler**

Compilers bridge source programs in high-level languages with the underlying hardware. A compiler requires

1) determining the correctness of the syntax of programs

2) generating correct and efficient object code

3) run-time organization

4) formatting output according to assembler and/or linker conventions.

A compiler consists of three main parts: the frontend, the middle-end, and the backend.

The **front end** checks whether the program is correctly written in terms of the programming language syntax and semantics.

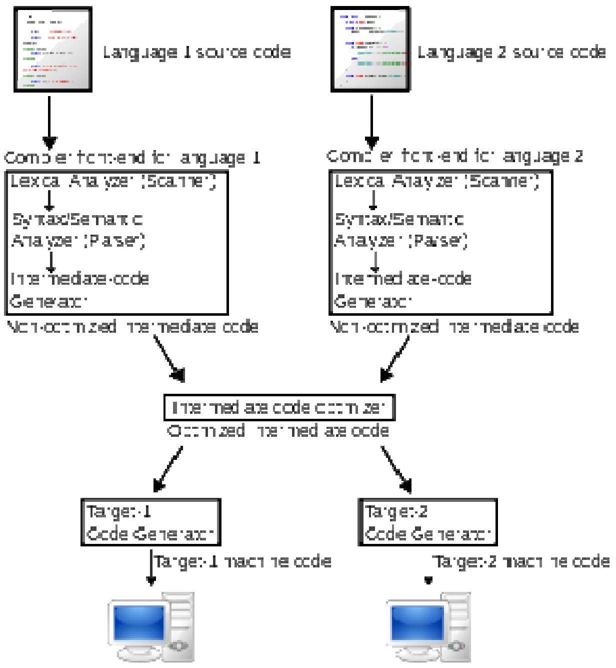
In this:-

1) legal and illegal programs are recognized.
2) Errors are reported.
3) Type checking is also performed by collecting type information.

The frontend then generates an *intermediate representation* or *IR* of the source code for processing by the middle-end.

The **middle end** is where optimization takes place. Typical transformations for optimization are removal of useless or unreachable code, discovery and propagation of constant values, relocation of computation to a less frequently executed place (e.g., out of a loop), or specialization of computation based on the context. The middle-end generates another IR for the following backend. Most optimization efforts are focused on this part.

The **back end** is responsible for translating the IR from the middle-end into assembly code. The target instruction(s) are chosen for each IR instruction. Register allocation assigns processor registers for the program variables where possible. The backend utilizes the hardware by figuring out how to keep parallel execution units busy, filling delay slots, and so on.


Language 1 source code
Language 2 source code
Compiler front-end for language 1
Compiler front-end for language 2
Lexical Analyzer (Scanner)
Syntax/Semantic Analyzer (Parser)
Intermediate-code Generator
Lexical Analyzer (Scanner)
Syntax/Semantic Analyzer (Parser)
Intermediate-code Generator
Non-optimized intermediate code
Non-optimized intermediate code
Intermediate code optimizer
Optimized intermediate code
Target-1 Code Generator
Target-2 Code Generator
Target-1 machine code
Target-2 machine code

A diagram of the operation of a typical multi-language, multi-target compiler

**3. Phases of compiler:-**

**3.1. Lexical analysis:-**

Lexical analysis is the process of analyzing a stream of individual characters, into a sequence of lexical tokens to feed into the parser. It separates characters of the source language into groups that logically belong together,these groups are knowns as tokens.The normal tokens are keywords such as DO or IF, identifiers and operator symbols.It is also known as scanner.

For example given the input string:

integer aardvark := 2, b;
Output of the lexical analyzer is:-
keyword integer
word aardvark
assignment operator
integer 2
comma
word b
semi_colon

**3.2. Syntax analysis:-**

The syntax analyzer groups tokens together into syntactic structures.It is also known as parser.it has two functions:-

1) It checks that the tokens appearing in its input occur in patterns that are permitted by the specification of the source language.
2) It imposes on the tokens a tree-like structure that is used by the subsequent phases of the compiler.

Syntax trees are the primary structure used for compilation, code analysis, binding, refactoring, IDE features, and code generation. No part of the source code is understood without it first being identified and categorized into one of many well-known structural language elements.

Syntax trees have three key attributes. The first attribute is that syntax trees hold all the source information in *full fidelity*. This means that the syntax tree contains every piece of information found in the source text, every grammatical construct, every lexical token, and everything else in between including whitespace, comments, and preprocessor directives.

For example, each literal mentioned in the source is represented exactly as it was typed.

The syntax trees also represent errors in source code when the program is incomplete or malformed, by representing skipped or missing tokens in the syntax tree.

This enables the second attribute of syntax trees. A syntax tree obtained from the parser is completely round trippable back to the text it was parsed from. From any syntax node, it is possible to get the text representation of the sub-tree rooted at that node. This means that syntax trees can be used as a way to construct and edit source text. By creating a tree you have by implication created the equivalent text, and by editing a syntax tree, making a new tree out of changes to an existing tree, you have effectively edited the text.

The third attribute of syntax trees is that they are *immutable* and thread-safe. This means that after a tree is obtained, it is a snapshot of the current state of the code, and never changes. This allows multiple users to interact with the same syntax tree at the same time in different threads without locking or duplication. Because the trees are immutable and no modifications can be made directly to a tree, factory methods help create and modify syntax trees by creating additional snapshots of the tree. The trees are efficient in the way they reuse underlying nodes, so the new version can be rebuilt fast and with little extra memory.

A syntax tree is literally a tree data structure, where non-terminal structural elements parent other elements. Each syntax tree is made up of *nodes*, *tokens*, and *trivia.*

Syntax nodes are one of the primary elements of syntax trees. These nodes represent syntactic constructs such as declarations, statements, clauses, and expressions. .

All syntax nodes are non-terminal nodes in the syntax tree, which means they always have other nodes and tokens as children. As a child of another node, each node has a *parent* node that can be accessed through the Parent property. Because nodes and trees are immutable, the parent of a node never changes.

Syntax tokens are the terminals of the language grammar, representing the smallest syntactic fragments of the code. They are never parents of other nodes or tokens. Syntax tokens consist of keywords, identifiers, literals, and punctuation. For example, an integer literal token represents a numeric value.

**3.3. Intermediate code generation:-**

The intermediate code generation phase transforms the parse tree which is the output of the syntax analyzer into an intermediate-language representation of the source program. One popular type of intermediate language is three address code. A three address code statement is:

A := B op C

Where A,B and C are operands and op is a binary operator.

**3.4. Code optimization:-**

Code optimization is an optional phase designed to improve the intermediate code so that the ultimate program runs faster and takes less space. Its output is another intermediate code program that does the same job as the original, but in a way that saves time and space. Optimization is the process of transforming a piece of code to make more efficient (either in terms of time or space) without changing its output or side-effects. The only difference visible to the code's user should be that it runs faster and/or consumes less memory. It is really a misnomer that the name implies you are finding an "optimal" solution— in truth, optimization aims to improve, not perfect, the result.

Optimization can occur at a number of "levels":

**3.4.1.Design level**

At the highest level, the design may be optimized to make best use of the available resources.

**3.4.2.Source code level**

Avoiding poor quality coding can also improve performance, by avoiding obvious "slowdowns". After that, however, some optimizations are possible that actually decrease maintainability.

**3.4.3.Build level**

Between the source and compile level, directives and build flags can be used to tune performance options in the source code and compiler respectively.

**3.4.4.Compile level**

Use of an optimizing compiler tends to ensure that the executable program is optimized at least as much as the compiler can predict.

**3.4.5.Assembly level**

At the lowest level, writing code using an assembly language, designed for a particular hardware platform can produce the most efficient and compact code if the programmer takes advantage of the full repertoire of machine instructions.

**3.5. Code generation:-**

code generation is the process by which a compiler's code generator converts some intermediate representation of source code into a form that can be readily executed by a machine. It is a mechanism to produce the executable form of computer programs, such as machine code, in some automatic manner.

compiler's "code generation" phase include:

- Instruction selection: which instructions to use.
- Instruction scheduling: in which order to put those instructions. Scheduling is a speed optimization that can have a critical effect on pipelined machines.
- Register allocation: the allocation of variables to processor registers
- Debug data generation if required so the code can be debugged.

**Refernces:-**

1. Compiler textbook references A collection of references to mainstream Compiler Construction Textbooks
2. Aho, Alfred V.; Sethi, Ravi; and Ullman, Jeffrey D., *Compilers: Principles, Techniques and Tools*(ISBN 0-201-10088-6) link to publisher
3. Karen Ng, Principal Lead Program Manager, Microsoft Corporation Matt Warren, Principal Architect, Microsoft Corporation Peter Golde, Partner Architect, Microsoft Corporation Anders Hejlsberg, Technical Fellow, Microsoft Corporation*September 2012*