**INTRODUCTION**

A compiler is a computer program (or set of programs) that transforms source code written in a programming language (the source language) into another computer language (the target language, often having a binary form known as object code). The most common reason for wanting to transform source code is to create an executable program.

The name "compiler" is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly 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 program that translates from a low level language to a higher level one is a decompiler. A program that translates between high-level languages is usually called a language translator, source to source translator, or language converter. A language rewriter is usually a program that translates the form of expressions without a change of language.

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. Program faults caused by incorrect compiler behavior can be very difficult to track down and work around; therefore, compiler implementors invest a lot of time ensuring the correctness of their software. The term compiler-compiler is sometimes used to refer to a parser generator, a tool often used to help create the lexer and parser.

**COMPILER**

**COMPILATION**

Compilers enabled the development of programs that are machine-independent. Before the development of FORTRAN (FORmula TRANslator), the first higher-level language, in the 1950s, machine-dependent assembly language was widely used. While assembly language produces more reusable and relocatable programs than machine code on the same architecture, it has to be modified or rewritten if the program is to be executed on different hardware architecture.

With the advance of high-level programming languages soon followed after FORTRAN, such as COBOL, C, BASIC, programmers can write machine-independent source programs. A compiler translates the high-level source programs into target programs in machine languages for the specific hardwares. Once the target program is generated, the user can execute the program.

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, and 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 *frontend* checks whether the program is correctly written in terms of the programming language syntax and semantics. Here legal and illegal programs are recognized. Errors are reported, if any, in a useful way. 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 *backend* is responsible for translating the IR from the middle-end into assembly code. The target instruction(s) are chosen for each IR instruction. Variables are also selected for the registers. Backend utilizes the hardware by figuring out how to keep parallel FUs busy, filling delay slots, and so on. Although most algorithms for optimization are in NP, heuristic techniques are well-developed.

**Front End**

The front end analyzes the source code to build an internal representation of the program, called the intermediate representation or IR. It also manages the symbol table, a data structure mapping each symbol in the source code to associated information such as location, type and scope. This is done over several phases, which includes some of the following:

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. The topdown, recursive-descent, table-driven parsers used in the 1960s typically read the source one character at a time and did not require a separate tokenizing phase. Atlas Autocode, and Imp (and some implementations of ALGOL and Coral 66) are examples of stropped languages which compilers would have a Line Reconstruction phase.
2. Lexical analysis breaks the source code text into small pieces called tokens. Each token is a single atomic unit of the language, for instance a keyword, identifier or symbol name. The token syntax is typically a regular language, so a finite state automaton constructed from a regular expression can be used to recognize it. This phase is also called lexing or scanning, and the software doing lexical analysis is called a lexical analyzer or scanner.
3. Preprocessing. Some languages, e.g., C, require a preprocessing phase which supports macro substitution and conditional compilation. Typically the preprocessing phase occurs before syntactic or semantic analysis; e.g. in the case of C, the preprocessor manipulates lexical tokens rather than syntactic forms. However, some languages such as Scheme support macro substitutions based on syntactic forms.
4. Syntax analysis involves parsing the token sequence to identify the syntactic structure of the program. This phase typically builds a parse tree, which replaces the linear sequence of tokens with a tree structure built according to the rules of a formal grammar which define the language's syntax. The parse tree is often analyzed, augmented, and transformed by later phases in the compiler.
5. Semantic analysis is the phase in which the compiler adds semantic information to the parse tree and builds the symbol table. This phase performs semantic checks such as type checking (checking for type errors), or object binding (associating variable and function references with their definitions), or definite assignment (requiring all local variables to be initialized before use), rejecting incorrect programs or issuing warnings. Semantic analysis usually requires a complete parse tree, meaning that this phase logically follows the parsing phase, and logically precedes the code generation phase, though it is often possible to fold multiple phases into one pass over the code in a compiler implementation.

**Middle End**

The middle end performs optimizations on the intermediate representation in order to improve the performance and the quality of the produced machine code. The middle end contains those optimizations that are independent of the CPU architecture being targeted.

The main phases of the middle end include the following:

Analysis: This is the gathering of program information from the intermediate representation derived from the input; data-flow analysis is used to build use-define chains, together with dependence analysis, alias analysis, pointer analysis, escape analysis, etc. Accurate analysis is the basis for any compiler optimization. The control flow graph of every compiled function and the call graph of the program and are usually also built during the analysis phase. Optimization: the intermediate language representation is transformed into functionally equivalent but faster (or smaller) forms. Popular optimizations are inline expansion, dead code elimination, constant propagation, loop transformation and even automatic parallelization. Compiler analysis is the prerequisite for any compiler optimization, and they tightly work together. For example, dependence analysis is crucial for loop transformation.

The scope of compiler analysis and optimizations vary greatly, from as small as a basic block to the procedure/function level, or even over the whole program (interprocedural optimization). Obviously, a compiler can potentially do a better job using a broader view. But that broad view is not free: large scope analysis and optimizations are very costly in terms of compilation time and memory space; this is especially true for interprocedural analysis and optimizations.

Interprocedural analysis and optimizations are common in modern commercial compilers from HP, IBM, SGI, Intel, Microsoft, and Sun Microsystems. The open source GCC was criticized for a long time for lacking powerful interprocedural optimizations, but it is changing in this respect. Another open source compiler with full analysis and optimization infrastructure is Open64, which is used by many organizations for research and commercial purposes.

Due to the extra time and space needed for compiler analysis and optimizations, some compilers skip them by default. Users have to use compilation options to explicitly tell the compiler which optimizations should be enabled.

**Back End**

The term back end is sometimes confused 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.

The main phases of the back end include the following:

1. Analysis: This is the gathering of program information from the intermediate representation derived from the input. Typical analyses are data flow analysis to build use-define chains, dependence analysis, alias analysis, pointer analysis, escape analysis etc. Accurate analysis is the basis for any compiler optimization. The call graph and control flow graph are usually also built during the analysis phase.

2. Optimization: the intermediate language representation is transformed into functionally equivalent but faster (or smaller) forms. Popular optimizations are inline expansion, dead code elimination, constant propagation, loop transformation, register allocation and even automatic parallelization.

3. Code generation: the transformed intermediate language is translated into the output language, usually the native machine language of the system. This involves resource and storage decisions, such as deciding which variables to fit into registers and memory and the selection and scheduling of appropriate machine instructions along with their associated addressing modes.

**ARCHITECTURE OF COMPILER**

A compiler can broadly be divided into two phases based on the way they compile.

**Analysis** **Phase**

Known as the **front-end** of the compiler, the analysis phase of the compiler reads the source program, divides it into core parts, and then checks for lexical, grammar, and syntax errors. The analysis phase generates an intermediate representation of the source program and symbol table, which should be fed to the Synthesis phase as input.

Front end

Password

Back end

Password

Intermediate code representation

Machine code

Password

Source code

Password

Synthesis

Analysis

Password

**Synthesis Phase**

Known as the **back-end** of the compiler, the synthesis phase generates the target program with the help of intermediate source code representation and symbol table. A compiler can have many phases and passes.

 Pass: A pass refers to the traversal of a compiler through the entire program.

 Phase: A phase of a compiler is a distinguishable stage, which takes input from the previous stage, processes and yields output that can be used as input for the next stage. A pass can have more than one phase.

**PHASES** **OF** **COMPILER**

The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler. Let us understand the phases of a compiler.

**1. Lexical Analysis**

The first phase of scanner works as a text scanner. This phase scans the source code as a stream of characters and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens as:

<token-name, attribute-value>

**2. Syntax Analysis**

The next phase is called the syntax analysis or parsing. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar, i.e., the parser checks if the expression made by the tokens is syntactically correct.

**3. Semantic Analysis**

Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not, etc. The semantic analyzer produces an annotated syntax tree as an output.

**4. Intermediate Code** **Generation**

After semantic analysis, the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

**5. Code Optimization**

The next phase does code optimization of the intermediate code. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).

**6. Code Generation**

In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language. The code generator translates the intermediate code into a sequence of (generally) re-locatable machine code. Sequence of instructions of machine code performs the task as the intermediate code would do.

**7. Symbol Table**

It is a data-structure maintained throughout all the phases of a compiler. All the identifiers’ names along with their types are stored here. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it. The symbol table is also used for scope management.

**COMPILER OUTPUT**

One classification of compilers is by the platform on which their generated code executes. This is known as the target platform. A native or hosted compiler is one which output is intended 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 often used when developing software for embedded systems that are not intended to support a software development environment. The output of a compiler that produces code for a virtual machine (VM) may or may not be executed on the same platform as the compiler that produced it. For this reason such compilers are not usually classified as native or cross compilers.

**CONCLUSION**

Compiler technology is useful for a more general class of applications. Many programs share the basic properties of compilers: they read textual input, organize it into a hierarchical structure and then process the structure. An understanding how programming language compilers are designed and organizedcanmakeiteasiertoimplementthesecompilerlikeapplicationsas well. Using a high-level language for programming has a large impact on how fast programs can be developed**.** Compared to machine language, the notation used by programming languages is closer to the way humans think about problems, The compiler can spot some obvious programming mistakes, Programs written in a high-level language tend to be shorter than equivalent programs written in machine language. The *same* program can be compiled to many different machine languages and, hence, be brought to run on many different machines.

**REFERENCES**

* Compiler textbook references (http://www.informatik.uni-trier.de/~ley/db/books/compiler/index.html) A collection of references to mainstream Compiler Construction Textbooks.
* Aho, Alfred V.; Sethi, Ravi; and Ullman, Jeffrey D., Compilers: Principles, Techniques and Tools ISBN 0-201-10088-6 link to publisher (http://www.aw.com/catalog/academic/product/0,4096,0201100886,00.html). Also known as “The Dragon Book.”
* Allen, Frances E., "A History of Language Processor Technology in IBM" (http://www.research.ibm.com/journal/rd/25 5/ibmrd2505Q.pdf), IBM Journal of Research and Development, v.25, no.5, September 1981. Allen, Randy; and Kennedy, Ken, Optimizing Compilers for Modern Architectures, Morgan Kaufmann Publishers, 2001. ISBN 1-55860-286-0
* Appel, Andrew Wilson