Volume: 2 Issue: X Month of publication: October 2014
DOI:

International Journal for Research in Applied Science & Engineering Technology(IJRASET)

Compiler Designing

Abhishek Jain[1], Ashish Chauhan[2], Manjeet Saini[3]

Department of Computer Science and Engineering, Dronacharya College of Engineering

Khentawas, Farrukh Nagar – 123506, Gurgaon, Haryana

Abstract: Need of advancement in techniques of interpretation and run time compilation increases day by day to handle the complexities of daily life problems. In order to reduce the complexity of designing and building computers advancement in designing of compilers increases. All of these compilers are made to execute the simple commands very quickly. A program for a computer is built by combining these simple commands into a program in what is known as machine language. This is a most tedious and error prone process of development such kind of program so we have high level programming languages for this purpose but they do not work like machine language. Therefore to fill the different between them, we require a program interface known as compiler.

Keywords: Compiler, interpreter, cross-compilers, JVM, .obj files, .exe files, intermediate, byte-code, machine-independent, Assembler, translator, dis-assembler, de-compiler.

I. INTRODUCTION

A compiler translates the high level program that is comfortable to programmers into machine level that is required by the computer to execute the instructions at the hardware level. During this process compiler also helps to fix the errors in the program occurred at runtime. Compilers enabled the development of programs that are machine-independent. Before the development of FORTRAN, the first higher-level language, in the 1950s, machine-dependent assembly language was widely used. While assembly language produces more abstraction than machine code on the same architecture, just as with machine code, it has to be modified or rewritten if the program is to be executed on different computer hardware architecture. With the advent of high-level programming languages that followed FORTRAN, such as COBOL, C, and BASIC, programmers could write machine-independent source programs. A compiler translates the high-level source programs into target programs in machine languages for the specific hardware. Once the target program is generated, the user can execute the program.

An interpreter is a Program that directly executes or performs, instructions written in a programming or scripting language, without previously compiling them into a machine language program.

An interpreter generally uses one of the following strategies for program execution:

1. Parse the Source code and perform its behavior directly
2. translate source code into some efficient intermediate representation and immediately execute this
3. Explicitly execute stored precompiled code, made by a compiler which is part of the interpreter system.

In traditional compilation, the executable output of the linkers (.exe files or .dll files or a library) is typically reloadable when run under a general operating system, much like the object code modules are but with the difference that this relocation is done dynamically at run time, i.e. when the program is loaded for execution.

Assembly language is a type of low-level language and a program that compiles it is more commonly known as an assembler, with the inverse program known as a dis-assembler.

A program that translates from a low level language to a higher level one is a de-compiler.

A program that translates between high-level languages is usually called a language translator, source to source translator, language converter, or language rewriter. The last term is usually applied to translations that do not involve a change of language.

A program that translates into an object code format that is not supported on the compilation machine is called a cross compiler and is commonly used to prepare code for embedded applications.

A. Need of Compiler and Interpreter are as follows:-

1) Compared to machine language, the notation used by programming languages is closer to the way humans think about problems.
2) The compiler can spot some obvious programming mistakes.
3) Programs written in a high-level language tend to be shorter than equivalent programs written in machine language
4) Up to some extend compiler, interpreter and high level language make program execution machine independent because via the use of them we can translates the same program written into high level for many machines.
5) Some-times programs written into high level language does not meet the speed execution therefore some critical codes written into either machine language or low level language. One of the characteristics of a good compiler is that it executes the code similar to the speed of machine language.
6) A compiler can thus make almost all the conversions from source code semantics to the machine level once and for all (i.e. until the program has to be changed) while an interpreter has to do some of this conversion work every time a statement or function is executed.

B. Types of Compilers and Interpreters:-

1) Byte code interpreters:-

There is a spectrum of possibilities between interpreting and compiling, depending on the amount of analysis performed before the program is executed. For example, Emacs Lisp is compiled to byte-code, which is a highly compressed and optimized representation of the Lisp source, but is not machine code (and therefore not tied to any particular hardware). This "compiled" code is then interpreted by a byte-code interpreter (itself written in C). The compiled code in this case is machine code for a virtual machine, which is implemented not in hardware, but in the byte-code interpreter. The same approach is used with the Forth code used in Open Firmware systems: the source language is compiled into "F code" (a byte-code), which is then interpreted by a virtual machine. Control tables - that do not necessarily ever need to pass through a compiling phase - dictate appropriate algorithmic control flow via customized interpreters in similar fashion to byte-code interpreters.

2) Abstract Syntax Tree interpreters

In the spectrum between interpreting and compiling, another approach is transforming the source code into an optimized Abstract Syntax Tree (AST) then executing the program following this tree structure, or using it to generate native code Just-In-Time. In this approach, each sentence needs to be parsed just once. As an advantage over byte-code, the AST keeps the global program structure and relations between statements (which is lost in a byte-code representation), and when compressed provides a more compact representation. Thus, using AST has been proposed as a better intermediate format for Just-in-time compilers than byte-code. Also, it allows performing better analysis during runtime.

3) Just-in-time compilation:-

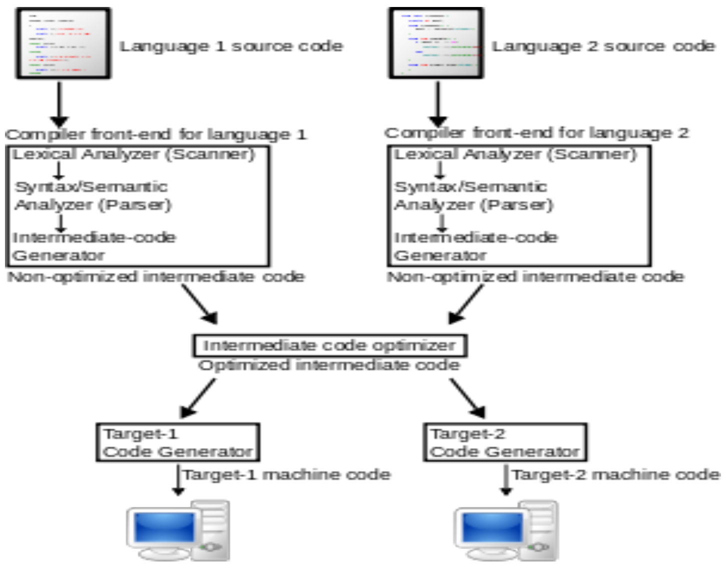
Further blurring the distinction between interpreters, byte-code interpreters and compilation is just-in-time compilation (or JIT), a technique in which the intermediate representation is compiled to native machine code at runtime. This confers the efficiency of running native code, at the cost of startup time and increased memory use when the byte-code or AST is first compiled. Adaptive optimization is a complementary technique in which the interpreter profiles the running program and compiles its most frequently executed parts into native code. Both techniques are a few decades old, appearing in languages such as Smalltalk in the 1980s.

4) Self-interpreter:-

A self-interpreter is a programming language interpreter written in a programming language which can interpret itself; an example is a BASIC interpreter written in BASIC. Self-interpreters are related to self-hosting compilers.

If no compiler exists for the language to be interpreted, creating a self-interpreter requires the implementation of the language in a host language (which may be another programming language or assembler). By having a first interpreter such as this, the system is bootstrapped and new versions of the interpreter can be developed in the language itself. It was in this way that Donald Knuth developed the TANGLE interpreter for the language WEB of the industrial standard TeX typesetting system.

Clive Gifford introduced a measure quality of self-interpreter (the Eigen ratio), the limit of the ratio between computer time spent running a stack of N self-interpreters and time spent to run a stack of N−1 self-interpreters as N goes to infinity. This value does not depend on the program being run.


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

II. COMPILER DESIGNING

A. Hardware compilation:-

The output of some compilers may target computer hardware at a very low level, for example a Field Programmable Gate Array (FPGA) or structured Application-specific integrated circuit (ASIC).] Such compilers are said to be hardware compilers or synthesis tools because the source code they compile effectively controls the final configuration of the hardware and how it operates; the output of the compilation is not instructions that are executed in sequence - only an interconnection of transistors or lookup tables. For example, XST is the Xilinx Synthesis Tool used for configuring FPGAs. Similar tools are available from Altera, Synplicity, Synopsys and other vendors.

B. One-pass versus multi-pass compilers:-

Classifying compilers by number of passes has its background in the hardware resource limitations of computers. Compiling involves performing lots of work and early computers did not have enough memory to contain one program that did all of this work. So compilers were split up into smaller programs which each made a pass over the source (or some representation of it) performing some of the required analysis and translations.

The ability to compile in a single pass has classically been seen as a benefit because it simplifies the job of writing a compiler and one-pass compilers generally perform compilations faster than multi-pass compilers.

In some cases the design of a language feature may require a compiler to perform more than one pass over the source.

The disadvantage of compiling in a single pass is that it is not possible to perform many of the sophisticated optimizations needed to generate high quality code. It can be difficult to count exactly how many passes an optimizing compiler makes. For instance, different phases of optimization may analyze one expression many times but only analyze another expression once.

While the typical multi-pass compiler outputs machine code from its final pass, there are several other types:

1) A "source-to-source compiler" is a type of compiler that takes a high level language as its input and outputs a high level language. For example, an automatic parallelizing compiler will frequently take in a high level language program as an input and then transform the code and annotate it with parallel code annotations (e.g. Open MP) or language constructs (e.g. Fortran's DO ALL statements)

2) Just-in-time compiler, used by Smalltalk and Java systems, and also by Microsoft .NET's Common Intermediate Language (CIL)

3) Applications are delivered in byte-code, which is compiled to native machine code just prior to execution.

C. Structure of a compiler:

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

1) The front end: Verifies syntax and semantics, and generates an intermediate representation or IR of the source code for processing by the middle-end. Performs type checking by collecting type information. Generates errors and warning, if any, in a useful way. Aspects of the front end include lexical analysis, syntax analysis, and semantic analysis.

2) The middle end: Performs optimizations, including 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. Generates another IR for the backend.

3) The back end: Generates the assembly code, performing register allocation in process. (Assigns processor registers for the program variables where possible.) 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.

D. Front end:

The compiler frontend 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.

While the frontend can be a single monolithic function or program, as in a scanner less parser, it is more commonly implemented and analyzed as several phases, which may execute sequentially or concurrently. This is particularly done for good engineering: modularity and separation of concerns. Most commonly today this is done as three phases: Lexing, parsing, and semantic analysis. Lexing and parsing comprise the syntactic analysis (word syntax and phrase syntax, respectively), and in simple cases these modules (the lexer and parser) can be automatically generated from a grammar for the language, though in more complex cases these require manual modification or writing by hand. The lexical grammar and phrase grammar are usually context-free grammars, which simplifies analysis significantly, with context-sensitivity handled at the semantic analysis phase. The semantic analysis phase is generally more complex and written by hand, but can be partially or fully automated using attribute grammars. These phases themselves can be further broken down – Lexing as scanning and evaluating, parsing as first building a concrete syntax tree (CST, parse tree), and then transforming it into an abstract syntax tree (AST, syntax tree).

In some cases additional phases are used, notably line reconstruction and preprocessing, but these are 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. The top-down, 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 Auto code, 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. This may not be a separate step – it can be combined with the parsing step in scanner less parsing, in which case parsing is done at the character level, not the token level.

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.

E. 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 (see also Sethi-Ull-man algorithm). Debug data may also need to be generated to facilitate debugging.

Compiler analysis is the prerequisite for any compiler optimization, and they tightly work together. For example, dependence analysis is crucial for loop transformation.

In addition, 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 (inter-procedural 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 inter-procedural analysis and optimizations.

Inter-procedural 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 inter-procedural 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.

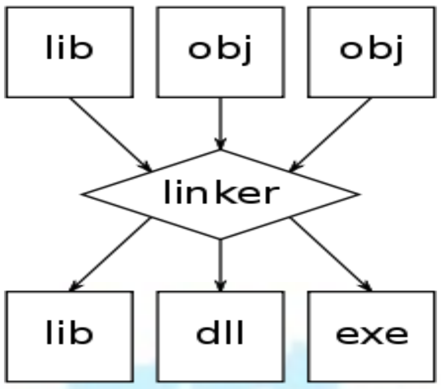
Due to the extra time and space needed for compiler analysis and optimizations, some compilers skip them by default. Users have to use compilation options to explicitly tell the compiler which optimizations should be enabled.

III. DEVELOPMENT CYCLE

During the software Development cycle, programmers make frequent changes to source code. When using a compiler, each time a change is made to the source code, they must wait for the compiler to translate the altered source files and link all of the binary code files together before the program can be executed. The larger the program, the longer the wait. By contrast, a programmer using an interpreter does a lot less waiting, as the interpreter usually just needs to translate the code being worked on to an intermediate representation (or not translate it at all), thus requiring much less time before the changes can be tested. Effects are evident upon saving the source code and reloading the program. Compiled code is generally less readily debugged as editing, compiling, and linking are sequential processes that have to be conducted in the proper sequence with a proper set of commands.

IV. LINKING PROCESS

Programs written in a high level language are either directly executed by some kind of interpreter or converted into machine code by a compiler (and assembler and linker) for the CPU to execute.



The linking process:-

Object files and static libraries are assembled in to a new library or executable.

While compilers (and assemblers) generally produce machine code directly executable by computer hardware, they can often (optionally) produce an intermediate form called object code. This is basically the same machine specific code but augmented with a symbol table with names and tags to make executable blocks (or modules) identifiable and re-allocate. Compiled programs will typically use building blocks (functions) kept in a library of such object code modules. A linker is used to combine (pre-made) library files with the object file(s) of the application to form a single executable file. The object files that are used to generate an executable file are thus often produced at different times, and sometimes even by different languages

A simple interpreter written in a low level language (i.e. assembly) may have similar machine code blocks implementing functions of the high level language stored, and executed when a function's entry in a look up table points to that code. However, an interpreter written in a high level language typically use another approach, such as generating and then walking a parse tree, or by generating and executing intermediate software-defined instructions, or both.

Thus, both compilers and interpreters generally turn source code (text files) into tokens, both may (or may not) generate a parse tree, and both may generate immediate instructions (for a stack machine, quadruple code, or by other means). The basic difference is that a compiler system, including a (built in or separate) linker, generates a standalone machine code program, while an interpreter system instead performs the actions described by the high level program.

V. DISTRIBUTION

A compiler converts source code into binary instruction for a specific processor's architecture, thus making it less portable. This conversion is made just once, on the developer's environment, and after that the same binary can be distributed to the user's machines where it can be executed without further translation. A cross compiler can generate binary code for the user machine even if it has a different processor than the machine where the code is compiled.

An interpreted program can be distributed as source code. It needs to be translated in each final machine, which takes more time but makes the program distribution independent of the machine's architecture. However, the portability of interpreted source code is dependent on the target machine actually having a suitable interpreter. If the interpreter needs to be supplied along with the source, the overall installation process is more complex than delivery of a monolithic executable since the interpreter itself is part of what need be installed.

The fact that interpreted code can easily be read and copied by humans can be of concern from the point of view of copyright. However, various systems of encryption and obfuscation exist. Delivery of intermediate code, such as byte-code, has a similar effect to obfuscation, but byte-code could be decoded with a de-compiler or dis-assembler.

VI. CONCLUSION

Characteristic of a good compiler is that it can execute the instructions quickly as the assembly or machine level language. So, our aim is here to maximize the speed of the compiler and make the code machine independent so that it can be executed at large scale.

Like java we can make our compilers advance so that they are able to produce an intermediate code which is not compiled completely and stores with a different file and further provide it as input to an interpreter which is used to convert the intermediate code into the machine dependent code. Due to which we have to develop single compiler for all machine and different interpreter for different platform which perform the main task of compiling the source code into the machine executable code.

Like java JVM compiler vendors also provides some **cross compilers** which works Similar to JVM but in case of cross compiler the source code is directly converted into the binary executable code due to which for every different hardware we require a differ compiler and program is compiled repeatedly for different machine processors but in java once compiled and execute every time concept is used where intermediate code is formed by compiling source code and that code can be executed on different machine processors by their respective **JVM**’s no need to compile program repeatedly here.

VII. FUTURE SCOPE

Compilers have emerged smarter over the years, yet they still lack the point of employing Artificial Intelligence (AI) in an attempt to see the future.
Compilers operate with a knowledge that is justified by arguments of a certain kind; i.e., knowledge of the code beyond the language syntax definitions, and it’s not feasible to determine that in what way a person, file or database is going to interact with an application until it actually happens.

Compilers nowadays use the if-statement approach. The need of the hour is to move to a higher level of abstraction. Building a compiler to catch runtime and logic errors would save millions of developer hours per year.

REFERNECES

[1] Aho, Alfred V., Hopcroft, J. E., and Ullman, Jeffrey D. [1974]. The Design and Analysis of Computer Algorithms. Addision Wesley, Reading, MA.

[2] Aho, Alfred V. and Johnson, Stephen C. [1976]. Optimal code generation for expression trees. Journal of the ACM, 23(3):488{501.

[3] Baker, T. P. [1982]. A one-pass algorithm for overload resolution in Ada. ACM Transactions on Programming Languages and Systems, 4(4):601{614.

[4] Balzer, R. M. [1969]. EXDAMS - extendable debugging and monitoring system. In Spring Joint Computer Conference, volume 34 of AFIPS Conference Proceedings, pages 567{580. AFIPS Press, Montvale, NJ.

[5] Cichelli, R. J. [1980]. Minimal perfect hash functions made simple. Communications of the ACM, 23(1):17{19.

[6] Clark, D. W. and Green, C. C. [1977]. An empirical study of list structure in LISP. Communications of the ACM, 20(2):78{87.