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

Pioneering Compiler Design

NikhitaUpreti;Divya Bali&Aabha Sharma
CSE,Dronacharya College of Engineering, Gurgaon, Haryana, India

nikhita.upreti@gmail.comdivyabali16@gmail.com
aabha6@gmail.com

Abstract

Generally compiling is a term which is often heard by everyone who is associated with programming, even if remotely. This paper enlightens the structure of compiler, various phases and tools used for its construction. Compiler is a program which converts a high level language program/code into binary instructions (machine language) that our computer can interpret, understand and take the appropriate steps to execute the same.

In earlier time only machine dependent programming languages were used and hence any program which could be run on one machine could not run on any other as it was specific to that machine. When high level languages that is machine independent language were first invented in the 40s and 50s no compilers had been written. In fifties the first compiler was written by Grace Hopper. The FORTRAN team lead by John Backus at IBM introduced the 1st complete compiler in 1957.

A compiler in real context takes a string and outputs another string. This definition covers all manner of software which converts one string to anther such as text formatters which convert an input language into a printable output, programs which tend to convert among various file formats or different programming languages and also web browsers.

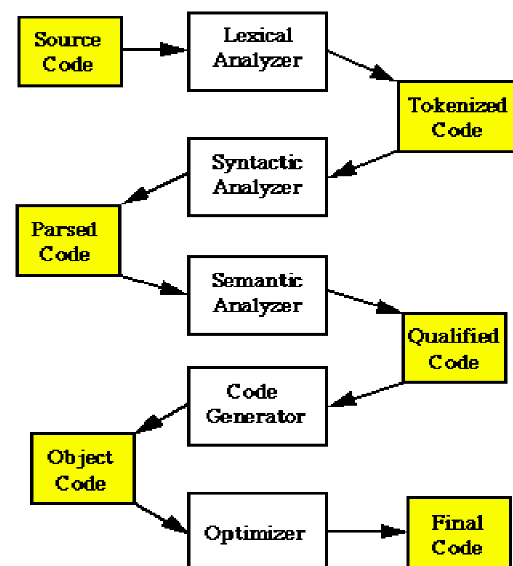
1. INTRODUCTION

A **compiler** is a computer program that implements a programming language specification to "translate" programs, usually as a set of files which constitute the *source code* written in*source language*, into their equivalent machine readable instructions **(the target language, often having a binary form known as object code**). This translation process is called *compilation*. We *compile* the source program to create the *compiled program*. The compiled program can then be run (or executed) to do what was specified in the original source program. The **source language** is always a higher-level language in comparison to machine code, written using some mixture of English words and mathematical notation, assembly language being the lowest compilable language (an *assembler* being a special case of a compiler that translates *assembly language* into machine code). Higher-level languages are the most complex to support in a compiler/interpreter, not only because they increase the level of abstraction between the source code and the resulting machine code, but because increased complexity is required to

formalize those abstract structures. The **target language** is normally a low-level language such as assembly, written with somewhat cryptic abbreviations for machine instructions, in this cases it will also run an assembler to generate the final machine code. But some compilers can directly generate machine code for some actual or virtual computer e.g. byte-code for the Java Virtual Machine. Another common approach to the resulting compilation effort is to target a *virtual machine*. That will do just-in-time compilation and byte-code interpretation and blur the traditional categorizations of compilers and interpreters. For example, C and C++ will generally be compiled for a target `architecture'. The draw-back is that because there are many types of processor there will need to be as many distinct compilations. In contrast Java will target a Java Virtual Machine, which is an independent layer above the 'architecture'. The difference is that the generated byte-code, not true machine code, brings the possibility of portability, but will need a Java Virtual Machine (the byte-code interpreter) for each platform. The extra overhead of this byte-code interpreter means slower execution speed.

A **translator** is a computer program that translates a program written in a given programming language into a functionally equivalent program in a different computer language, without losing the functional or logical structure of the original code (the "essence" of each program). These include translations between high-level and human-readable computer languages such as C++, Java and COBOL, intermediate-level languages such as Java bytecode, low-level languages such as assembler and machine code, and between similar levels of language on different computing platforms, as well as from any of these to any other of these. Arguably they also include translators between software implementations and hardware/ASIC microchip implementations of the same program, and from software descriptions of a microchip to the logic gates needed to build it. Examples of widely used types of computer languages translators include interpreters, compilers and decompilers, and assemblers and disassemblers.

2. Structure of a Compiler



The cousins of the compiler are

1. Preprocessor.
2. Assembler.
3. Loader and Link-editor.

Front End vs Back End of a Compilers. The phases of a compiler are collected into front end and back end.

The front end includes all analysis phases end the intermediate code generator.

The back end includes the code optimization phase and final code generation phase.

The front end analyzes the source program and produces intermediate code while the back end synthesizes the target program from the intermediate code.

A naive approach (front force) to that front end might run the phases serially.

1. Lexical analyzer takes the source program as an input and produces a long string of tokens.
2. Syntax Analyzer takes an out of lexical analyzer and produces a large tree.
3. Semantic analyzer takes the output of syntax analyzer and produces another tree.
4. Similarly, intermediate code generator takes a tree as an input produced by semantic analyzer and produces intermediate code.

Minus Points

- Requires enormous amount of space to store tokens and trees.
- Very slow since each phase would have to input and output to and from temporary disk

Remedy

- Use syntax directed translation to inter leaves the actions of phases.
- Compiler construction tools.

Parser Generators:

The specification of input based on regular expression. The organization is based on finite automation.

Scanner Generator:

The specification of input based on regular expression. The organization is based on finite automation.

Syntax-Directed Translation:

It walks the parse tee and as a result generate intermediate code.

Automatic Code Generators:

Translates intermediate rampage into machine language.

Data-Flow Engines:

It does code optimization using data-flow analysis.

3.<u>**Phases of Compiler**</u>

3.1 Lexical Analyzer

- The main task is to read the input characters and produce as output sequence of tokens that the parser uses for syntax analysis.

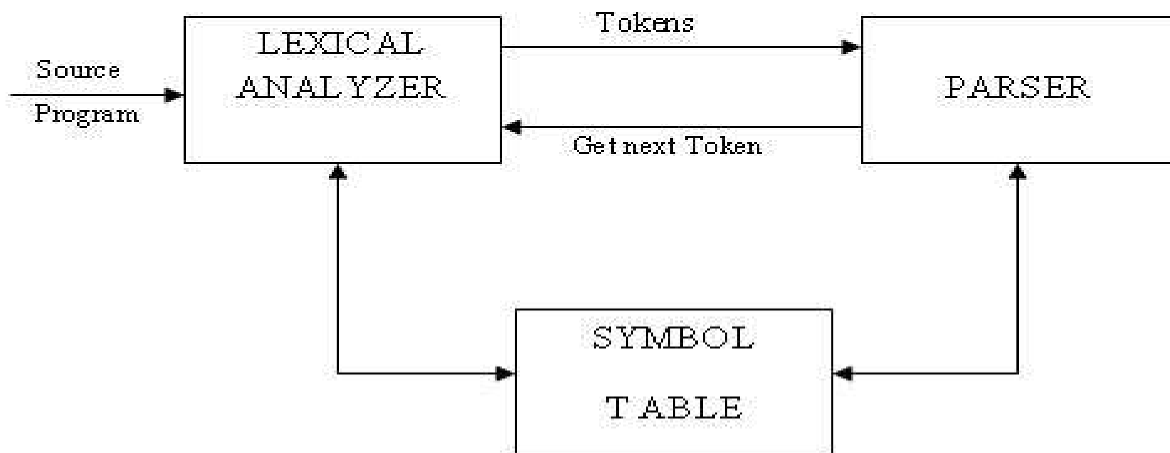


Fig 2.1 role of the lexical analyzer diagram

- Up on receiving a "get next token" command from the parser, the lexical analyzer reads input characters until it can identify the next token.
- Its secondary tasks are,
- One task is stripping out from the source program comments and white space is in the form of blank, tab, new line characters.
- Another task is correlating error messages from the compiler with the source program.
- Sometimes lexical analyzer is divided in to cascade of two phases.
  1) Scanning 2) lexical analysis.
- The scanner is responsible for doing simple tasks, while the lexical analyzer proper does the more complex operations.

- **Lexical Analyzer** reads the source program character by character and returns the *tokens* of the source program.

- A *token* describes a pattern of characters having same meaning in the source program. (such as identifiers, operators, keywords, numbers, delimeters and so on)

  Ex: newval :=oldval + 12 => tokens:

newval identifier
:=assignment operator
oldval identifier
\+ add operator
12 a number

- Puts information about identifiers into the symbol table.

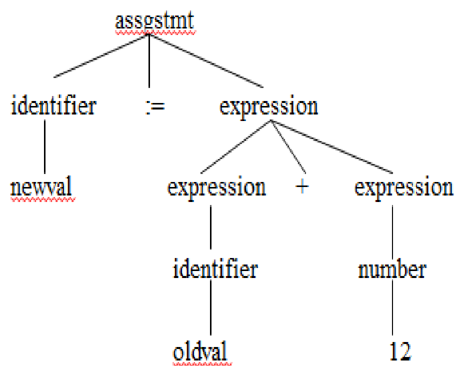
- Regular expressions are used to describe tokens (lexical constructs).

- A (Deterministic) Finite State Automaton can be used in the implementation of a lexical analyzer.

3.2. Syntax Analyzer

A **Syntax Analyzer** creates the syntactic structure (generally a parse tree) of the

given program.A **parse tree** describes a syntactic structure.



This is alternatively known as parsing. It is roughly the equivalent of checking that some ordinary text written in a natural language (e.g. English) is grammatically correct (without worrying about meaning).

The purpose of syntax analysis or parsing is to check that we have a valid sequence of tokens. Tokens are valid sequence of symbols, keywords, identifiers etc. Note that this sequence need not be meaningful; as far as syntax goes, a phrase such as "**true** + 3" is valid but it doesn't make any sense in most programming languages.

The parser takes the tokens produced during the lexical analysis stage, and attempts to build some kind of in-memory structure to represent that input. Frequently, that structure is an 'abstract syntax tree' (AST).

The parser needs to be able to handle the infinite number of possible valid programs that may be presented to it. The usual way to define the language is to specify a *grammar*. A grammar is a set of rules (or *productions*) that specifies the syntax of the language (i.e. what is a valid sentence in the language).

3.3Syntax Analyzer versus Lexical Analyzer

Which constructs of a program should be recognized by the lexical analyzer, and which ones by the syntax analyzer?

- Both of them do similar things; But the lexical analyzer deals with simple non-recursive constructs of the language.
- The syntax analyzer deals with recursive constructs of the language.
- The lexical analyzer simplifies the job of the syntax analyzer.
- The lexical analyzer recognizes the smallest meaningful units (tokens) in a source program.
- The syntax analyzer works on the smallest meaningful units (tokens) in a source program to recognize meaningful structures in our programming language.

3.4 Semantic Analysis

- Semantic analysis is applied by a compiler to discover the meaning of a program by analyzing its parse tree or abstract syntax tree.
- A program without grammatical errors may not always be correct program.
  - *pos = init + rate * 60*
  - What if *pos* is a class while *init* and *rate* are integers?

- ☐ This kind of errors cannot be found by the parser
- ☐ Semantic analysis finds this type of error and ensure that the program has a meaning.

- ■ **Static semantic checks** (done by the compiler) are performed at compile time
  - ☐ Type checking
  - ☐ Every variable is declared before used
  - ☐ Identifiers are used in appropriate contexts
  - ☐ Check subroutine call arguments
  - ☐ Check labels
- ■ **Dynamic semantic checks** are performed at run time, and the compiler produces code that performs these checks
  - ☐ Array subscript values are within bounds
  - ☐ Arithmetic errors, e.g. division by zero
  - ☐ Pointers are not dereferenced unless pointing to valid object
  - ☐ A variable is used but hasn't been initialized
  - ☐ When a check fails at run time, an exception is raised

3.5 Intermediate Code Generation:

- The syntax and semantic analysis generate a explicit intermediate representation of the source program.
- The intermediate representation should have two important properties:
- It should be easy to produce,
- And easy to translate into target program.
- Intermediate representation can have a variety of forms.
- One of the forms is: three address code; which is like the assembly language for a machine in which every location can act like a register.
- Three address code consists of a sequence of instructions, each of which has at most three operands.

3.6 Code Optimization and Code Generation:

- Code optimization phase attempts to improve the intermediate code, so that faster-running machine code will result.
- The final phase of the compiler is the generation of target code, consisting normally of relocatable machine code or assembly code.
- Memory locations are selected for each of the variables used by the program.
- Then, the each intermediate instruction is translated into a sequence of machine instructions that perform the same task.

4. Compiler Construction Tools

- **Lex&Yacc**- The classic Unix tools for compiler construction.Lex is a "tokenizer," helping to generate programs whose control flow is directed by instances of regular expressions in the input stream. It is often used to segment input in preparation for further parsing (as with Yacc).

Yaccprovides a more general parsing tool for describing the input to a computer program. The Yacc user specifies the grammar of the input along with code to be invoked as each structure in that grammar is recognized. Yacc turns that specification into a subroutine to process the input.

If you are writing a compiler, that "process" involves generating code to be assembled to generate the object code. Alternatively, if you are writing an interpreter, the "code to be invoked" will be code controlling flow of the user's application.

- **Lemon** - A LALR(1) parser generator that claims to be faster and easier to program than Bison or Yacc.
- **GCC** - RTL Representation-Most of the work of the compiler is done on an intermediate representation called register transfer language. In this language, the instructions to be output are described, pretty much one by one, in an algebraic form that describes what the instruction does.

People frequently have the idea of using RTL stored as text in a file as an interface between a language front end and the bulk of GNU CC. This idea is not feasible. GNU CC was designed to use RTL internally only. Correct RTL for a given program is very dependent on the particular target machine. And the RTL does not contain all the information about the program.

- Zephyr Compiler Infrastructure
- The National Compiler Infrastructure Project
- The SUIF Compiler - Software Distribution
- TENDRA / ANDF -compilation tools

5. Applications of compiler techniques

- 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 andorganized can make it easier to implement these compiler like applications aswell
- More importantly, tools designed for compiler writing such as lexical analyser generators and parser generators can make it vastly easier to implement such applications
- Thus, compiler techniques - An important knowledge for computer scienceengineers
- Examples:
  - Document processing: Latex, HTML, XML
  - User interfaces: interactive applications, file systems, databases
  - Natural language treatment

6. Conclusion

Compiler is language processor used to translate program written in high level language into the machine level language. It is also use to cover the "GAP" between Humans and the computer language. A program written in high level programming language is called the source program. The source program is stored on the disk in a file. The compiler translates the source program into machine codes and makes another program file is called the object file. The object file contains the translated program. Files, source and object are saved on the diskpermanently.

The object programs translated by compiler can executed a number of times without translating it again. If there are any errors in the source program the compiler specifies the errors at the end of compilation. The errors must be removed before the compiler can successfully compile the source program.

Computer understands only two words 0 and 1. Machine language or binary languages were used to write compilers. But it is very difficult to write complex code in form of 0 and 1. So we use high level programming languages are used to write compiler. Compiler is also used to communicate with hardware.

References

[1] Kaur, A., &Manhas, R. (2008). Use of internet services and resources in the engineering colleges of Punjab and Haryana (India): a study. *The International Information & Library Review*, *40*(1), 10-20.

[2] Goswami, P., Bhatia, P. K., &Hooda, V. (2009). Effort estimation in Component Based Software Engineering. *International Journal of Information Technology and Knowledge Management*, *2*(2), 437-440.

[3] Khan, S. A., Harish, R., Shokat Ali, R., Jain, V., & Raj, N. Distributed shared memory-an overview.

[4] Yadav, Y., &Yadav, P. Virtual Local Area Network.