COMPILER CONSTRUCTION (INTRODUCTION) **Fall 2019**

Lecture 1-2: July 18-19, 2019

Instructor: K.R. Chowdhary *: Professor of CS*

Disclaimer: *These notes have not been subjected to the usual scrutiny reserved for formal publications. They may be distributed outside this class only with the permission of the Instructor.*

1.1 Objectives and Outcomes

The objective the compiler course is to understand the basic principles of compiler design, its various constituent parts, algorithms and data structures required to be used in the compiler.

Yet, other objective is, to understand relations between computer architecture and how its understanding is useful in design of a compiler.

The other objective is, how to construct efficient algorithms for compilers.

The outcome is to acquire basic skills for designing the compilers, as well as the knowledge of compiler design.

After this course a student will know, in some depth, how a compiler works. In particular, he/she should understand the structure of a compiler, and how the source and target languages influence various choices in its design.

It will give you a new appreciation for programming language features and the implementation challenges they pose, as well as for the actual hardware architecture and the run time system in which your generated code executes.

Understanding the details of typical compilation models will also make you a more distinct programmer.

You will also understand some specific components of compiler technology, such as lexical analysis, grammars and parsing, type-checking, intermediate representations, static analysis, common optimizations, instruction selection, register allocation, code generation, and runtime organization.

The knowledge gained in the subject should be broad enough that if you are confronted with the task of contributing to the implementation of a real compiler in the field, you should be able to do so confidently and quickly. For many of you, this will be the first time you have to write, maintain, and evolve a complex piece of software. You will have to program for correctness, while keeping an eye on efficiency, both for the compiler itself and for the code it generates. Because you will have to rewrite the compiler from lab to lab, and also because you will be collaborating with a partner, you will have to pay close attention to issues of modularity and interfaces.

1.2 Introduction

The programming languages are notations for people as well as for computers, as how the computations are performed. The advancement in information technology, we see today is due to computers, and programs running in them. The programs are written in programming languages. Before a program is run on any computer it is to be translated into machine language, of that particular hardware on which it runs, unless it is written already in the machine language. The software that do this translation is called *compiler*.

In the history of developments of compilers, what ever the standard techniques available today were known in 1970s. Since then, the general improvements in the compilers have been in the following areas:

- automation in significant part of the construction of compilers;
- improvement in the speed of the target code, due to optimizing algorithms;
- greater ability of compilers to be relatively easily re-targeted or written for new target machines languages, partly due to automation, and partly due to better understanding of the required compiler structure.

Improvements have come in special language features, like, exception handling, generics, object oriented features such as dynamic binding, and parallelization. The other improvement that has taken place is translation of functional languages, with new algorithms for type checking, type classes, and interpretation by tree reduction.

An important aspect of techniques of translators is the strong interaction with the language design. For example introduction of block-structured languages, like AlGOL and Pascal, have pushed up the stack-based translation algorithms. Consequently, the stack-based algorithms influenced the development of language semantics, such as the lexical scope rule, and the principle of syntax directed translation. In such translation, the semantics of the program, i.e., its execution behavior, is directed reflected in the syntax, or the structure, so that translation is guided by this structure. This rule could have been formulated as semantics-based-syntax.

The other aspect of the development has been the constant effort of computing community to constantly rediscover creative translation techniques. The was primarily due to lack of detailed documentation of specific translators in the computer science documentation. The lack of documentation was due to commercial reasons or proprietary translators.

In this course, we will discuss about how to design and implement a compiler. These principles of compiler design are applicable to many applications, which appeared particularly after arrival of Internet. The principles of compiler design are important in natural language processing, natural language translation, natural language understanding, Information Retrieval, etc.

The components of the hardware comprise CPU (central processing unit), memory, internal architecture of the CPU, bus architecture, and the instruction set of the machine. The final code generated by a compiler is in the the form of these machine instructions.

Thus, a compiler can be said to be a language processor, that reads as input a high level language program, called *source programs* S, and translates it and provides at its output the program in machine language, called *object program* T (see Fig. 1.1).

Figure 1.1: A compiler

Fig. 1.2 shows execution of the compiled (object code). The object code, when run on the hardware, it receives its input as data, and produces its output as result.

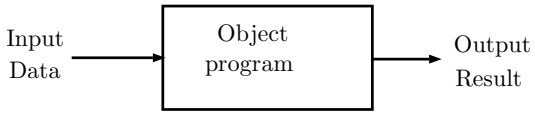

Figure 1.2: Executing an object program

Because a compiled program is to run on the hardware of the machine, therefore to understand the compiler, and its design it is important to understand the architecture/hardware of the computer also.

When the object program/machine language program is running on the hardware, each instruction of the machine program is *interpreted* by the hardware, that means, the functions specified in the instructions are carried out by the hardware. Therefore, the hardware is *interpreter* of machine language.

It is possible to do the interpretation of high level language also, where each of the high level instruction/statement is carried out directly, by a program (not the hardware), called *interpreter*. The term interpreter is usually used for this interpreter. The Fig. 1.3 shows the interpreter for a high level language.

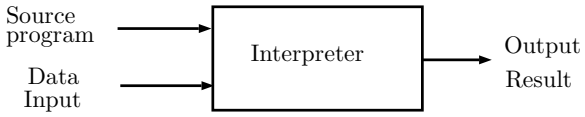

Figure 1.3: Interpreter for high level language

1.2.1 Compiler vs. Interpreter

From the above discussions, we understand that using a compiler we produce first, entire object program, and then run it on the hardware. Therefore, unless the translation is fully correct, it is not possible to produce the target/object program. However, the interpreter is different. In one sense, because, it does not translate, and in other, because it executes each of the high level language statement, one at a time, sequentially, until some error is encountered or the program ends after execution.

A Java language program's compilation is different from C. In the C, we first compile C program into object code, and then run it on hardware. Whereas in Java, first a java program is compiled into a code called *byte code*, which is different from machine code. Then as next step, this byte code is interpreted by an interpreter, called, *java virtual machine* (JVM). The java virtual machine is not hardware, but a program that runs on the hardware.

A compiler is usually faster than Interpreter, so there is performance penalty while using an interpreter. The advantage of interpreter is that compilation step is eliminated. For the languages like, C, C++, Fortran, there are compilers, but for the languages like, Prolog, and more dynamic languages, like, LISP, and SMALLTALK there are interpreters.

1.3 Language Processing System

All the languages have some common properties, like, languages have alphabets, alphabets make words, and words make sentences or statements. The only well defined alphabet sequences are words, and well defined word sequences are sentences. The sentences have definite structures, called *syntax* of the sentence, and meaning associated will each sub-unit of sentence, called meaning or *semantic* of the sub-unit of sentence. These are true for all the languages, whether it is a high level language, a human language (e.g., English), and machine/assembly language.

Irrespective of what language it is, all the language processing systems have some common features, as shown in the Fig. 1.4.

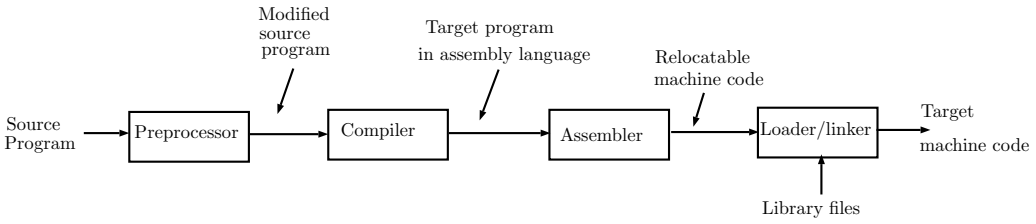

Figure 1.4: A high level language processing system

1.4 Compiler Structure

So far we have discussed that compiler is a black box, with input (source program) and output (machine or target code). Thus, we can say that compiler is a function c, that maps input set of statements S with the output set of statements T ($T = f_c(S)$). However, this mapping is not one-to-one, and of complex nature. Broadly speaking, there are two parts of this mapping: *analysis*, and *synthesis*.

The first, i.e., analysis part breaks the program into constituent pieces and imposes a grammatical (syntax) structure on them. This structure is used for generating an intermediate representation (IR) of the program. The analysis part checks that the program is as per the correct syntax (grammar), or semantically sound. If not, then errors are flagged to the user/programmer, so that the same can be corrected. One output of analysis part is intermediate representation (intermediate code) (see Fig. 1.5).

In addition, analysis part also called *front end* of the compiler) collects the information about the source program in a data structure called *symbol table*. This data structure is passed to the synthesis part of compiler along with the intermediate representation.

The synthesis part builds the target program (often in assembly language), using the intermediate representation and the symbol table. The synthesis part of the compiler is called

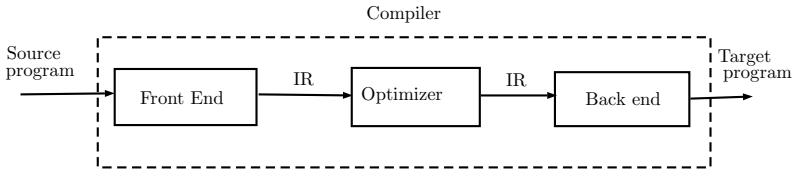

Figure 1.5: Major components of compiler

back end of the compiler.

1.5 Review Questions

1. What are the functions of a compiler?
2. Give names of any two compilers that run on Windows.
3. Give names of any two compilers that run on Linux.
4. How an interpreter is different from a compiler?
5. How a Java compiler and interpreter are different from conventional compilers and interpreters?
6. Give merits of compiler compared to interpreter.
7. Give merits of an interpreter compared to a compiler.
8. Which phase handles the type checking in compiler?
9. Which phase of the compiler handles the operations on index of any array?
10. What is JVM?

1.6 Exercises

1. While there are number of phases in a compiler, why it is preferred to have 2-3 passes only ?
2. There are number of other subjects in computer science curricula, how the compiler course is related to following courses?
   (a) data structure
   (b) discrete structures
   (c) operating systems
   (d) programming languages
   (e) graph theory
   (f) C language
3. What is relocatable code? Can you always load your executable code into any part of the memory? Justify for Yes/No.
4. What are the functions of linker? What actually it does?

References

[1] Compilers: Principles, Techniques, and Tools (2nd Edition) by Alfred V. Aho , Monica S. Lam, et al., Sep 10, 2006

[2] Compiler design in C (Prentice-Hall software series) by Allen I Holub, Jan 1, 1990

[3] Engineering a Compiler, by Keith D. Cooper and Linda Torczon, Morgan Kaufmann Publishers, 2004.