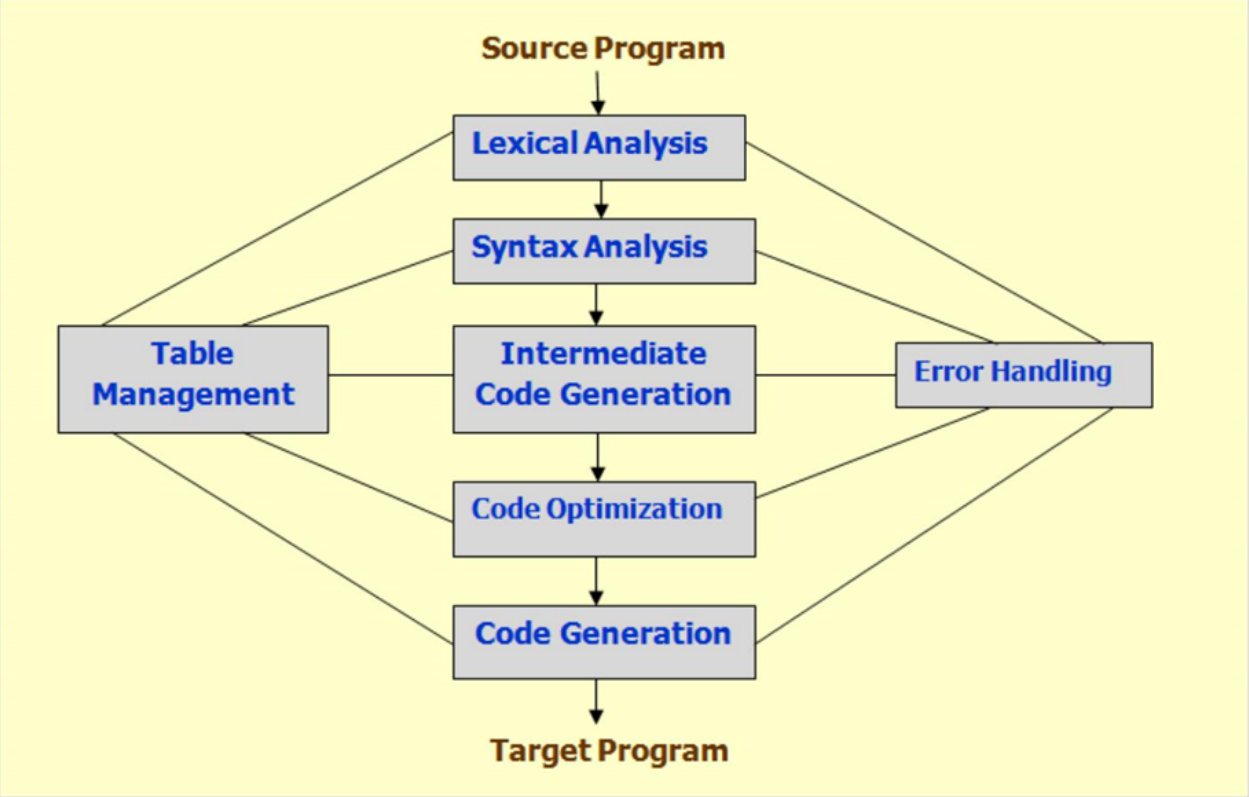
THE DUMMIES' GUIDE TO COMPILER DESIGN



Rosina S Khan

Dedicated to You,

The Valued Reader

Contents

Preface

While students in Computer Science and Engineering (CSE) field or any other equivalent field program in high-level languages and run their programs in editors using a compiler, they do need to understand the mysteries and complexities about how a compiler functions. And the best way to do that is to grasp the underlying principles and actually design a compiler in lab. This is where this book comes into the picture. In a simple, lucid way, the content of this book is made available to the students of CSE or any other equivalent program so that they can understand and grab all the concepts behind Compiler Design conveniently and thoroughly.

Now the principles and theory behind designing a compiler presented in this book are nothing new and they are presented as they have always been known but the real difference lies in the fact that they have been outlined in a really simple and easy-to understand way. Now I have collected some of the resources from varying sources and assembled with mine to make the flow of reading logical, comprehensible and easy to grasp.

Some of these resources are:

[1] Aho A. V., Lam M. S., Sethi R., Ullman J.D., *Compilers: Principles, Techniques & Tools*, Pearson Education, 2nd Ed., 2007

[2] Aho A. V., Ullman J. D., *Principles of Compiler Design*, Addison-Wesley/Narosa, Twenty-third Reprint, 2004

[3] Dr Fegarsas's Lecture Notes on *Compilers*, CSE, UTA

Organization

Let me now explain the organization of this book.

Chapter 1 is an introductory chapter explaining compilers, translators, their significances and structure of a compiler.

Chapter 2 illustrates lexical analyzers which take input from source programs and produce group of characters called tokens, how they work and function and finally their implementation introducing such concepts as regular expressions, nondeterministic finite automata and deterministic finite automata.

Chapter 3 covers syntactic analysis which groups tokens into syntactic structures such as expressions and statements. For this, we use concepts such as context free grammar, derivations and parse trees.

Chapter 4 continues with syntactic analysis further covering shift-reduce parsing, operator-precedence parsing, top-down parsing, recursive-descent parsing and predictive parsers.

Chapter 5 further continues with syntactic analysis, portraying a special kind of bottom-up parser, the LR parser which scans input from left to right and how they help with syntactic analysis.

Chapter 6 outlines syntax directed translation that introduces intermediate code generation, which is actually an extension of context-free grammars.

Chapter 7 mainly covers storage of variables within program code in a run-time stack.

Chapter 8 explains intermediate representation (IR) specification in areas of frame layout.

Chapter 9 portrays the role of a type checker in the design of a compiler.

Chapter 10 includes code optimization in order to improve the code space and time-wise before the final code generation.

Chapter 11 introduces code generation in machine language format, the final phase of a compiler.

There is also an Appendix at the very end outlining a miscellaneous exercise on compiler design on which students can work out throughout the whole semester in parallel with theory lectures.

Acknowledgments

I deeply acknowledge my heart-felt thanks to the authors and publisher of Compilers: Principles, Techniques & Tools and Principles of Compiler Design as well as Dr Fegaras, for using and collecting their resources and combining them with mine and hence the birth of this very book.

Last but not the least I am thankful to my family for all their support while writing out this book.

C H A P T E R 1

Introduction to Compilers

1.1 What actually is a Compiler?

A translator converts one program in a specific programming language as input to a program in another programming language as output. If the source language is a high-level language such as C++ or Java and the object or target language is assembly language or machine language, then such a translator is known as a compiler.

The function of the compiler takes place in two steps:

1) First, the source program is compiled or translated into the object program.
2) Second, the resulting object program is stored in memory and executed.

1.2 Other Translators

Certain translators transform a programming language into a less complex language called intermediate code, which can be executed directly with a program called interpreter. Here we can interpret the intermediate code acting as some sort of machine language.

Interpreters are smaller in size than compilers and help in the implementation of complex programming language structures. However, the main downside of interpreters is that they take more execution time than compilers.

Besides compilers, there are other translators as well. For instance, if the source language is assembly language and the target language is machine language, the translator is known as an assembler. As another instance, if a translator converts a high-level language to another high-level language, it is termed as a preprocessor.

1.3 What is the Significance of Translators?

We know machine languages are only sequences of 0's and 1's. If we program an algorithm in machine language, it not only becomes tedious but also becomes prone to making errors. All operations and operands must be numeric and therefore it becomes difficult to distinguish them, which is a serious downside. Another problem that arises is that it becomes inconvenient to modify them. So under these circumstances, machine languages are not reliable to start coding and this is exactly where the picture of high-level languages comes in.

A family of high-level languages has been invented so that the programmer can code in a way nearer to his thought processes, ideas and concepts rather than always think at the machine language level and code, which is almost always impossible. A step away from the machine language is the assembly language which uses mnemonic codes for both operations and data addresses. Thus, a programmer could write ADD X, Y in assembly language rather than use sequences of 0's and 1's for the same operation using machine language. However, the computer only understands machine language and so the assembly language needs to be translated to machine language and the translator which carries out this function is known as an assembler.

1.4 Macros

Macros are statements nearer to assembly language statements but different from them in that they use a single memory address along with the operation code. For example, our previous assembly code, ADD X, Y could be broken down into three macro operations, LOAD, ADD and STORE – all using single memory addresses as shown below:

```
MACRO       ADD2   X, Y
            LOAD   Y
            ADD    X
            STORE  Y
ENDMACRO
```

The first statement gives the name ADD2 to the macro along with its dummy arguments X, Y. The next three statements define the macro, assuming the machine has only one register, the Accumulator, the other name for Register A.

LOAD Y is equivalent to Y -> Acc

which means content of memory address Y is transferred to the Accumulator.

The next statement ADD X is equivalent to Acc + X -> Acc

which means content of the Accumulator is added to the content of memory address X and the result is stored in the Accumulator.

The third statement STORE Y is equivalent to Acc -> Y

which means the content of the Accumulator is stored in memory address Y.

In this way the assembly code ADD X, Y is broken into macro statements and the same Add operation happens in the latter case. This is useful when the number of registers in the machine is limited.

In the above code, we defined a macro. Now I shall explain how we can use a macro.

After definition of ADD2 X, Y, a macro use happens when we come across the statement ADD2 A, B. In ADD2 A, B statement, A and B replace X and Y respectively and is translated to:

LOAD B

ADD A

STORE B

Thus macro use is like a function call to the function definition as in a high-level language such as C, C# or Java.

1.5 High-Level Languages

A high-level programming language makes the programming task simpler, but it also introduces a problem. We now need a program to convert to a language the machine understands - in other words, the machine language. In that case, this program becomes a compiler similar to the assembler for an assembly language.

A compiler is more complex to write than an assembler. Sometimes compilers have appended with them an assembler so that the compiler produces assembly code initially, which is then assembled, loaded and converted into machine language.

1.6 The Structure of a Compiler

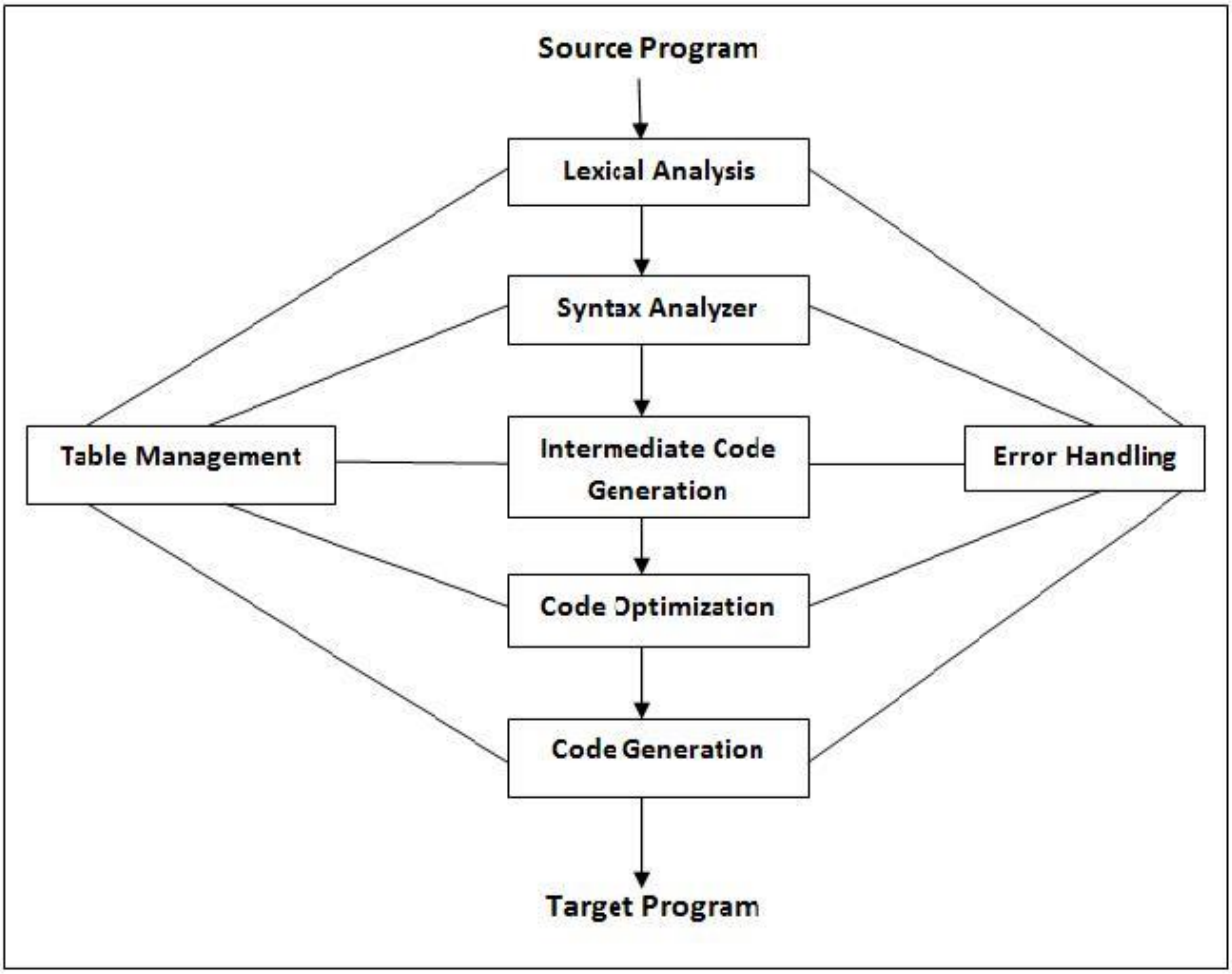


Fig. 1: Phases of a Compiler

As with our definition, a compiler converts a high-level language program into a machine language program. But the whole process does not occur in a single step but in a series of subprocesses called phases as shown in Fig. 1.

Each phase of the diagram becomes a chapter of this book because the phases of the compiler structure leads to the compiler design and that is what this book is about. Now let's go briefly through all the phases of the compiler.

The first phase of the compiler called lexical analyzer or scanner separates the characters of the source program into groups called tokens that logically belong together. Tokens can be keywords such as, DO or IF, identifiers or variables such as, X or NUM, operator symbols such as <= or + and punctuation symbols such as parenthesis or commas. The tokens can be represented by integer codes for instance, DO can have the integer code 1, '+' can have 2 while 'identifiers' 3.

The syntax analyzer or parser, the next phase of the compiler, groups tokens into syntactic structures. For instance, the three tokens in A+B can be grouped together to form a syntactic structure called an expression. Expressions can further be grouped to form statements. Sometimes the syntactic structure can be represented as a tree whose leaves form the tokens.

The output of syntax analyzer or parser is a stream of simple instructions called intermediate code. The intermediate code generator which produces the intermediate code in the third phase usually use instructions such as simple macros with one operator and a small number of operands. The macro ADD2 statement explained in section 1.4 is

a bright example of this. The main difference between intermediate code and assembly code is that the former does not need to specify registers for each operation unlike the latter.

Code optimization, the next phase after intermediate code generator, is an optional phase that is geared to improving the intermediate code so that the ultimate object program runs faster and/or takes less space.

The final phase, Code Generator, should be designed in such a way that it produces a truly efficient object code, which is challenging, both in practical and theoretical terms.

In table management or bookkeeping portion of the compiler, a data structure called symbol table keeps track of the names (identifiers) used by the program and records important information about each such as, its type whether integer, real etc.

The error handler handles errors as the information passes from the source program through one phase to another.

Both the table management and error handling routines interact with all phases of the compiler.