Draft version: September 14, 2018

Chapter 1 – Introduction

1.1 What is a compiler?

A **compiler** translates a program in a **source language** to a program in a **target language**. The most well known form of a compiler is one that translates a high level language like C into the native assembly language of a machine so that it can be executed. And of course there are compilers for other languages like C++, Java, C#, and Rust, and many others.

The same techniques used in a traditional compiler are also used in any kind of program that processes a language. For example, a typesetting program like TEX translates a manuscript into a Postscript document. A graph-layout program like Dot consumes a list of nodes and edges and arranges them on a screen. A web browser translates an HTML document into an interactive graphical display. To write programs like these, you need to understand and use the same techniques as in traditional compilers.

Compilers exist not only to *translate* programs, but also to *improve* them. A compiler assists a programmer by finding errors in a program at compile time, so that the user does not have to encounter them at runtime. Usually, a more strict language results in more compile-time errors. This makes the programmer's job harder, but makes it more likely that the program is correct. For example, the Ada language is infamous among programmers as challenging to write without compile-time errors, but once working, is trusted to run safety-critical systems such as the Boeing 777 aircraft.

A compiler is distinct from an **interpreter**, which reads in a program and then executes it directly, without emitting a translation. This is also sometimes known as a **virtual machine**. Languages like Python and Ruby are typically executed by an interpreter that reads the source code directly.

Compilers and interpreters are closely related, and it is sometimes possible to exchange one for the other. For example, Java compilers translate Java source code into Java **bytecode**, which is an abstract form of assembly language. Some implementations of the Java Virtual Machine work as interpreters that execute one instruction at a time. Others work by translating the bytecode into local machine code, and then running the machine code directly. This is known as **just in time compiling** or **JIT**.

1.2 Why should you study compilers?

You will be a better programmer. A great craftsman must understand his or her tools, and a programmer is no different. By understanding more deeply how a compiler translates your program into machine language, you will become more skilled at writing effective code and debugging it when things go wrong.

You can create tools for debugging and translating. If you can write a parser for a given language, then you can write all manner of supporting tools that help you (and others) debug your own programs. An integrated development environment like Eclipse incorporates parsers for languages like Java, so that it can highlight syntax, find errors without compiling, and connect code to documentation as you write.

You can create new languages. A surprising number of problems are made easier by expressing them compactly in a custom language. (These are sometimes known as **domain specific languages** or simply **little languages**.) By learning the techniques of compilers, you will be able to implement little languages and avoid some pitfalls of language design.

You can contribute to existing compilers. While it's unlikely that you will write the next great C compiler (since we already have several), language and compiler development does not stand still. Standards development results in new language features; optimization research creates new ways of improving programs; new microprocessors are created; new operating systems are developed; and so on. All of these developments require the continuous improvement of existing compilers.

You will have fun while solving challenging problems. Isn't that enough?

1.3 What's the best way to learn about compilers?

The best way to learn about compilers is to *write your own compiler* from beginning to end. While that may sound daunting at first, you will find that this complex task can be broken down into several stages of moderate complexity. The typical undergraduate computer science student can write a complete compiler for a simple language in a semester, broken down into four or five independent stages.

1.4 What language should I use?

Without question, you should use the C programming language and the X86 assembly language, of course!

Ok, maybe the answer isn't quite that simple. There is an ever-increasing number of programming languages that all have different strengths and weaknesses. Java is simple, consistent, and portable, albeit not high performance. Python is easy to learn and has great library support, but weak typing. Rust offers exceptional static type-safety, but is not (yet) widely used. It is quite possible to write a compiler in nearly any language, and you could use this book as a guide to do so.

However, we really think that you should learn C, write a compiler in C, and use it to compile a C-like language which produces assembly for a widely-used processor, like X86 or ARM. Why? Because it is important for you to learn the ins and outs of technologies that are in wide use, and not just those that are abstractly beautiful.

C is the most widely-used portable language for low-level coding (compilers, and libraries, and kernels) and it is also small enough that one can learn how to compile every aspect of C in a single semester. True, C presents some challenges related to type safety and pointer use, but these are manageable for a project the size of a compiler. There are other languages with different virtues, but none as simple and as widely used as C. Once you write a C compiler, then you are free to design your own (better) language.

Likewise, the X86 has been the most widely-deployed computer architecture in desktops, servers, and laptops for several decades. While it is considerably more complex than other architectures like MIPS or SPARC or ARM, one can quickly learn the essential subset of instructions necessary to build a compiler. Of course, ARM is quickly catching up as a popular architecture in the mobile, embedded, and low power space, so we have included a section on that as well.

That said, the principles presented in this book are widely applicable. If you are using this as part of a class, your instructor may very well choose a different compilation language and different target assembly, and that's fine too.

1.5 How is this book different from others?

Most books on compilers are very heavy on the abstract theory of scanners, parsers, type systems, and register allocation, and rather light on how the design of a language affects the compiler and the runtime. Most are designed for use by a graduate survey of optimization techniques.

This book takes a broader approach by giving a lighter dose of optimization, and introducing more material on the process of engineering a compiler, the tradeoffs in language design, and considerations for interpretation and translation.

You will also notice that this book doesn't contain a whole bunch of fiddly paper-and-pencil assignments to test your knowledge of compiler algorithms. (Ok, there are a few of those in Chapters 3 and 4.) If you want to test your knowledge, then write some working code. To that end, the exercises at the end of each chapter ask you to take the ideas in the chapter, and either explore some existing compilers, or write parts of your own. If you do all of them in order, you will end up with a working compiler, summarized in the final appendix.

1.6 What other books should I read?

For general reference on compilers, I suggest the following books:

- **Charles N. Fischer, Ron K. Cytron, and Richard J. LeBlanc Jr, "Crafting a Compiler", Pearson, 2009.**
  This is an excellent undergraduate textbook which focuses on object-oriented software engineering techniques for constructing a compiler, with a focus on generating output for the Java Virtual Machine.

- **Christopher Fraser and David Hanson, "A Retargetable C Compiler: Design and Implementation", Benjamin/Cummings, 1995.**
  Also known as the "LCC book", this book focuses entirely on explaining the C implementation of a C compiler by taking the unusual approach of embedding the literal code into the textbook, so that code and explanation are intertwined.

- **Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, "Compilers: Principles, Techniques, and Tools", Addison Wesley, 2006.** *Affectionately known as the "dragon book", this is a comprehensive treatment of the theory of compilers from scanning through type theory and optimization at an advanced graduate level.*

Ok, what are you waiting for? Let's get to work.