A Very Simple Compiler Book

Philip W. Howard

2019

Contents

Preface

Given the existence of other very good books suitable for a course on compilers, why did I choose to write another one? There are several reasons:

1. Most compilers books seem to be written from the perspective that the course is preparing students to actually write a production compiler. That is not the perspective of the compiler courses I've taught. While the students in my course write a complete compiler, I readily acknowledge that very few of my students will work on a compiler as part of their career (although some have). The value of the course has more to do with helping the students to become better developers than in preparing them for a particular field. I have found that this difference in perspective alters what material I find most suitable for the courses I teach.

2. I find the price of most text books almost criminal. By writing my own text, I can make electronic copies freely available and make printed copies available for a reasonable price.

Part I

Overview

Chapter 1

Introduction

1.1 Why?

The first question that many prospective compiler students have is, "Why should I study compilers?" The following answers come to mind:

1. They're cool. Why wouldn't you want to study compilers?

2. It gives you a chance to understand what your primary tool does. It would seem strange for a carpenter to not know what a hammer does, or to not know the trade-offs between a hammer and a nail gun. Granted, a carpenter doesn't need to know the physics of how these tools work, but they should have a general idea.

   Many software developers never consider their compilers. They have this large blob of software known as an "IDE". They type specialized text into their IDE and click various buttons, and if they are luck, they get a working program. My claim is that better programmers have a better understanding of their tools. This allows them to make better use of these tools, and thus better use of their time, and thus (sometimes) more of their employer's money.

3. It gives you a chance to manage a large code base. I've heard of undergraduate computer science students who graduate having never managed anything bigger than a two week lab. In my ten week compiler course, students write an entire compiler composed of tens of source files and thousands of lines of code. Design and implementation decisions made early have strong consequences later in the term. This is a

valuable experience that every developer should have before they step into a job claiming they know what they're doing.

Another advantage of the large project is that it allows the introduction of testing techniques that are harder to make "real" in smaller environments.

4. Applied data structures. A compiler has a lot of data that needs to be managed. Many data structures present "clean" data structures, but real-life data-problems often aren't so clean. How about a cyclic tree that contains links to a stack of hash tables that in turn has links back to nodes of the tree? OK, if your tree is cyclic, you probably have a bug, but most data structures classes don't give you the opportunity to make (and then fix) such bugs. Having the opportunity to get buried in data is good for you.

5. Applied algorithms. I started to learn calculus in high school. I continued my learning of calculus in college. But I didn't really understand what a dx was until I encountered them in my calculus based physics class. They were no longer merely symbols on a paper, they were now applied, and that's how I finally grasped what they really were. In the same sense, a compiler allows you to apply quite a number of algorithms (and data structures). In doing so, they should be more "real".

6. Performance considerations. Computers have gotten so fast, we rarely have to worry about performance (time or memory) considerations. Most compiler runs are also fast enough that the compiler's performance doesn't much matter. But given a compiler, you can compile something very big like the Linux Kernel. Performance now matters. Because of this, a compiler class gives the right to talk about performance considerations in a practical sense.

1.2 What is a compiler?

Perhaps before asking, "Why study compilers?" it would have been best to establish what a compiler is. But most students, by the time they get to a compiler course, have already used compilers so they have some sense of what they are: The software that translates my source code into executable code. We need to be a bit more formal as well as a bit broader in our definition.

In the broadest sense, a compiler takes code as input and outputs an improved version of that code. For the compiler to be valid, it must follow two guiding principles:

1. The compiler must preserve the meaning of the original program.
2. The compiler must improve the code in some discernible way.

Figure 1.1: A compiler is a program that takes code as input and outputs an improved version of that code.

To illustrate the first principle, if you had a program that output the numbers from 1..10 and the compiler transformed it so that it output one of e.e. cummings's poems, then clearly the compiler did not preserve the meaning of the original program.

The second principle is a bit less clear. If the code is a program, and you want to execute your program, then transforming your source code into an executable program would be an improvement. But suppose your source was a C program and the output was an equivalent Java program. Would that be an improvement?

This definition of a compiler is deliberately broad. A C preprocessor (often a separate program from the C compiler) is considered a compiler. LaTeX, which was used to typeset this book, is a compiler. Indeed, any converter program can be considered a compiler given this definition. For the sake of this book (and most compiler courses), we will primarily restrict ourselves to compilers where the input is a programming language. The output, however, does not need to be executable code. It can be any form of "improved" coded.

Chapter 2

Compiler Structure

Compilers are typically broken into two broad pieces: the front-end and the back-end. The frond-end deals with language specific issues. The back-end deals with target machine specifics.

A C compiler would need a very different front-end compared with a Haskell compiler because the syntax of the languages is very different. One way to view the front-end of the compiler is that it answers the question, "Is this a valid program?" If you feed a Scheme program into an R compiler, the answer would certainly by "No". If a Java program that was missing a semicolon was fed into a Java compiler, the answer would also be, "No", but for very different reasons.

The back-end of a compiler deals with target specific issues. The back-end for a compiler that targets the Java Virtual Machine (JVM) would be different than the back-end for a compiler that targets and ARM processor. Both of those would be different from a C preprocessor that simply outputs text.

Separating the front-end from the back-end makes sense because they are logically quite different, but it also makes sense because if they are properly designed, you can mix and match different front-ends and back-ends. Suppose you had a really good C compiler, but you wanted to port it to target a different machine. There would be no reason to rewrite the front-end. The C language is the same no matter what machine you are targeting. So if you wrote a new back-end and attached it to your existing front-end, you would now have a new compiler. Mixing can go the other way as well. Starting with the same really good C compiler, suppose you wanted a compiler for a different language (such as Go). There is no reason to rewrite the back-end,

simply create a front-end for the new language and attach it to your existing back-end, and you have a new compiler.

For this mixing and matching to work seamlessly, there needs to be a common interface between the front-ends and the back-ends. This interface is typically called an Intermediate Representation (IR) — see Chapter 3. The front-end translates the language into the IR, and the back-end translates the IR to the target representation. This is illustrated in Figure 2.1

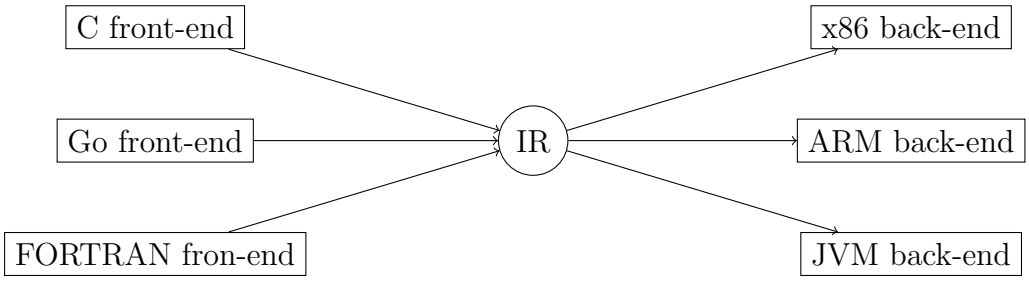

Figure 2.1: A properly structured compiler allows you to have multiple language-specific front-ends joined to multiple target-specific back-ends. The two a joined using an intermediate representation.

Compiler front-ends typically contain two components: A scanner (aka tokenizer or lexer), and a parser. Scanners are discussed in detail in Part II and parsers are discussed in Part III.

Most computer program source code comes in the form of a stream of characters (a text file). The scanner turns the stream of characters into a stream of tokens. For example, it the scanner read the characters "`some_variable_name`", it would output the single token `IDENTIFIER`. For a longer example, the string "`int main() { printf( "hello world"\n); }`" would be converted to the tokens "`TYPE IDENTIFIER ( ) { IDENTIFIER ( STRING_LIT ) ; }`".

Chapter 3

Intermediate Representations

Part II

Scanning

Chapter 4

Regular Expressions

This chapter discusses regular expressions, what they are for, and how to use them.

Chapter 5

Creating a Scanner from Regular Expressions

This chapter covers the algorithms for converting a list of Regular Expressions into code that can process them.

5.1 Thompson's

This section presents Thompson's construction.

5.2 Subset Construction

This section presents the Subset Construction

Chapter 6

Scanner Code

6.1 table driven

Here is a table driven scanner

6.2 switch statement

Here is a table drive scanner

Chapter 7

Automatically Generated Scanners

here is a description of flex.

Part III

Parsing

Chapter 8

Context Free Grammars

Chapter 9

Top-down recursive-descent parsers

Chapter 10

Bottom-up parsers

Chapter 11

Automatically generated parsers

Description of bison.

Part IV

The Back-end

Chapter 12

The Visitor Pattern

Chapter 13

Code generation

Chapter 14

Optimization

Subsubsection

This is just some text under a subsubsection.

Subsubsubsection This is just some text under a subsubsubsection.

Subsubsubsubsection This is just some text under a subsubsubsubsection.

- Bullet item 1
- Bullet item 2

Description List Each description list item has a term followed by the description of that term.

Bunyip Mythical beast of Australian Aboriginal legends.

14.1 Theorem-Like Environments

The following theorem-like environments (in alphabetical order) are available in this style.

Acknowledgement 1 *This is an acknowledgement*

Algorithm 2 *This is an algorithm*

Axiom 3 *This is an axiom*

Case 4 *This is a case*

Claim 5 *This is a claim*

Conclusion 6 *This is a conclusion*

Condition 7 *This is a condition*

Conjecture 8 *This is a conjecture*

Corollary 9 *This is a corollary*

Criterion 10 *This is a criterion*

Definition 11 *This is a definition*

Example 12 *This is an example*

Exercise 13 *This is an exercise*

Lemma 14 *This is a lemma*

Proof. This is the proof of the lemma. ■

Notation 15 *This is notation*

Problem 16 *This is a problem*

Proposition 17 *This is a proposition*

Remark 18 *This is a remark*

Summary 19 *This is a summary*

Theorem 20 *This is a theorem*

Proof of the Main Theorem. This is the proof. ■

Appendix A

The First Appendix

The `\appendix` command should be used only once. Subsequent appendices can be created using the Chapter command.

Appendix B

The Second Appendix

Some text for the second Appendix.

This text is a sample for a short bibliography. You can cite a book by making use of the command `\cite{KarelRektorys}`: [1]. Papers can be cited similarly: [2]. If you want multiple citations to appear in a single set of square brackets you must type all of the citation keys inside a single citation, separating each with a comma. Here is an example: [2, 3, 4].

Bibliography

[1] Rektorys, K., *Variational methods in Mathematics, Science and Engineering*, D. Reidel Publishing Company, Dordrecht-Hollanf/Boston-U.S.A., 2th edition, 1975

[2] Bertóti, E.: *On mixed variational formulation of linear elasticity using nonsymmetric stresses and displacements*, International Journal for Numerical Methods in Engineering., **42**, (1997), 561-578.

[3] Szeidl, G.: *Boundary integral equations for plane problems in terms of stress functions of order one*, Journal of Computational and Applied Mechanics, **2**(2), (2001), 237-261.

[4] Carlson D. E.: *On Günther's stress functions for couple stresses*, Quart. Appl. Math., **25**, (1967), 139-146.

Afterword

The back matter often includes one or more of an index, an afterword, acknowledgements, a bibliography, a colophon, or any other similar item. In the back matter, chapters do not produce a chapter number, but they are entered in the table of contents. If you are not using anything in the back matter, you can delete the back matter TeX field and everything that follows it.