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.

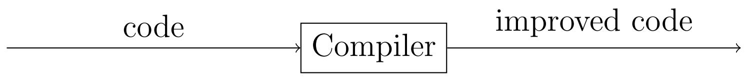


Figure 1.1: A compiler is a program that takes code as input and outputs an improved version of that code.

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

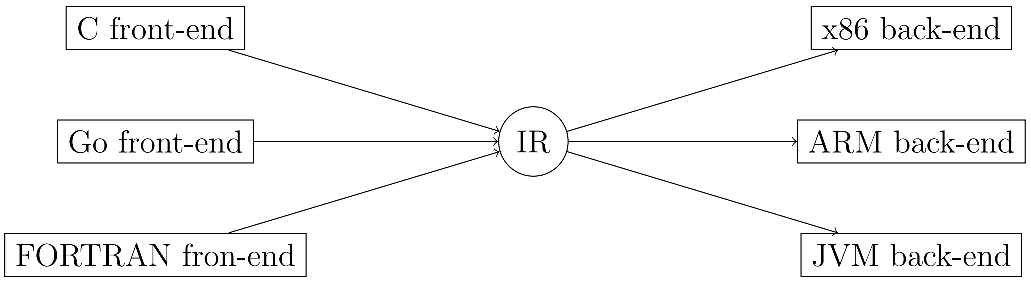


Figure 2.1: A properly structured compiler allows you to have multiple language-specific front-ends joined to multiple target-specific back-ends. The two a joined using an intermediate representation.

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 ) ; }`".

Note: Programmers normally format their code to make it "pretty". The code in the above examples was not formatted to emphasize that to a compiler, code is just a stream of characters.

The parser takes a stream of tokens and decides whether that stream of tokens makes up a valid program (more specifically a “compilation unit”) in the language. The parser only looks at the syntax of the language. As a result, “valid program” only means “a program that is syntactically correct”. It does not mean “a bug free program”.

Chapter 3

Intermediate Representations

Part II

Scanning

Chapter 4

Regular Expressions

A scanner turns a stream of characters into a stream of tokens. To facilitate this conversion, most compilers make use of regular expressions to define tokens. Many programmers are familiar with regular expressions from non-compiler contexts. Examples include using an asterisk (*) as a wildcard in a file name, or specifying patterns for the `grep` utility. Different programs use different syntax for specifying regular expressions. In this section a minimalist syntax for all regular expressions is presented. Most programs that interpret regular expressions enhance this syntax in various ways to make writing regular expressions easier, but the added syntax does not add extra capabilities.

Regular expressions include the following features:

Concatenation	Concatenation is gluing two strings end-to-end. For example, concatenating "`ab`" with "`bc`" yields the string "`abcd`".
Alternation	Alternation means to choose exactly one from a set of alternatives. Regular expressions use the vertical bar (`|`) to mean alternation. So the expression `a	b	c` means to choose either an '`a`', a '`b`', or a '`c`'.
Grouping	Parenthesis can be used for grouping operations much as they can in algebraic expressions.

Kleene Closure Kleene Closure means to take zero or more instances of a string. Kleene Closure is denoted by an asterisk (`*`). So, for example, ttx* means zero or more 'x' characters. Kleene Closure has higher precedence that concatenation so that `ab*` means `a(b*)` not `(ab)*`.

In addition to these operations, the Λ symbol is used to represent an empty string (a string with no characters in it).

The most common enhancements to this syntax are as follows:

zero or one The question mark (`?`) indicates zero or one of an item so that `a?` means the same as (Λ `| a`).

one or more The plus sign (`+`) is similar to Kleene Closure, but it is one-or-more not zero-or-more so that `a+` means the same as `aa*`.

character range Square brackets (`[]`) can be used to specify a character range so that `[a-m]` means any single character in the range '`a`' through '`m`'.

If we want a regular expression for integer constants, we could try

```
[0-9]+
```

but this allows any number of leading zeros. A better expression would be:

```
[1-9][0-9]*
```

This fixes the leading zero problem, but it doesn't allow the number zero. This can be fixed as follows:

```
0+([1-9][0-9]*)
```

If we want to allow negative numbers, we could add an optional minus sign:

```
0+(-?[1-9][0-9]*)
```

Exercises

1. Write a regular expression for a string containing any odd number of the letter `a`.
2. Write a regular expression for C (or Java) variable names. Valid characters include upper and lower case letters, digits, and the underscore (_).
3. Write a regular expression for a string containing any number (including zero) of a positive even number of `a`'s followed by an add number of `b`'s. The following are valid strings: `aaaabbb`, `aabaabaabbb`, `aaaaaabbbbbaabaabaab`. The following are not valid strings: `aaab`, `aaaabbbaa`, `bbbaab`.
4. For the previous question, state why each of the non-valid strings are non-valid.
5. Write a regular expression for a floating-point constant. The following rules apply:
   (a) The integer part cannot have leading zeros unless the integer part is zero.
   (b) If there is a decimal point, it must be followed by at least one digit.
   (c) The decimal part must not have trailing zeros unless the decimal part is zero.

Chapter 5

Creating a Scanner from Regular Expressions

To specify a scanner for a compiler, a developer could specify a regular expression for each token type. But how do you go from that specification to code that can actually process the input? This chapter discusses the algorithms used to convert regular expressions into a scanner. The general process is to convert the regular expression into a finite automaton. The finite automaton can then be evaluated using a simple table look-up on each character.

5.1 Finite Automata

Finite Automata (aka State Machines) consist of a finite number of states and transitions between states. One state is defined as the start state, and any number of states can be defined as final states. They are often drawn as diagrams as illustrated in Figure 5.1.

The start state in an FA is signified by an inbound arrow that doesn't originate from another state. Processing always starts in the start state. Final states are signified by a double circle. Transitions are labeled by the letter that is used to move from one state to another.

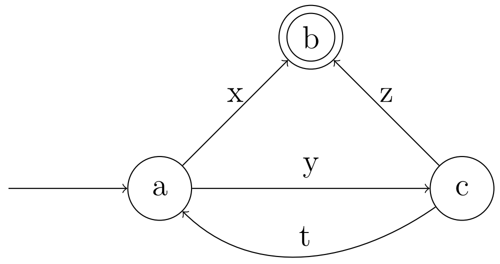


Figure 5.1: This is a sample FA. Node *a* is the start state (it has an incoming arrow from nowhere). State *b* is a final state (signified by the double-circle). The transitions are labeled by what letter is used to move from one state to another.

Using the FA in Figure 5.1, and the input *ytx*, the processing starts in State *a* (the start state). The *y* is used to transition to State *c*. The *t* is used to transition back to State *a*. The *x* is used to transition to State *b*. Since the input is exhausted while in a final state, the string is accepted by the FA.

Two conditions can cause a string to be rejected:

1. If the input is exhausted and the current state isn't a final state. The input *yty* illustrates this case.
2. If there is no outbound transition on the current letter. The input *yx* illustrates this case.

Given these rules for processing FA's, it should be clear that the FA in Figure 5.1 is equivalent to the regular expression `(yt)* (x | yz)`

5.1.1 Non-deterministic Finite Automota

The FA illustrated in Figure 5.1 is a Deterministic Finite Automaton (DFA). It is deterministic in the sense that for each character that is processed, the FA has exactly one choice on what to do. There is another class of FA's known as Non-deterministic Finite Automata (NFA's). With NFA's, for a given input, there is the potential for multiple choices on what to do. The choices can take two forms:

1. Multiple outbound edges labeled with the same letter. If that letter is read, any of the outbound edges labeled with that letter can be taken. This is illustrated by the edges from Node a labeled z in Figure 5.2.

2. Edges labeled Λ. These edges can be taken without consuming an input character. Node a has a Λ transition meaning you can leave Node a without consuming any characters.

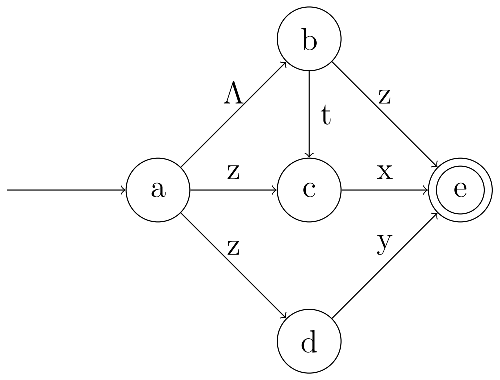


Figure 5.2: This is a sample NFA. Node a has multiple outbound transitions on z. It also has an outbound transition on Λ meaning you could leave Node a without consuming any characters.

The NFA in Figure 5.2 accepts the following stings:

z	This string is accepted by following the Λ transition to Node b and then using the z to transition to Node e.
tx	This string is accepted by following the Λ transition to Node b and then using the t to transition to Node c and the x to transition to Node e.
zx	This string is accepted by using the z to transition to Node c and then using the x to transition to Node e.
zy	This string is accepted by using the z to transition to Node d and then using the y to transition to Node e.

NFA's aren't any more powerful than DFA's–anything you can do with an NFA you can also do with a DFA[1]. The reason for introducing NFA's is that converting from a regular expression to code that accepts strings matching that regular expression makes use of NFA's.

5.2 Thompson's

The first step in converting a regular expression to executable code is to convert it to an equivalent NFA. For this conversion, we are going to use Thompson's Construction[2]. The beauty of Thompson's construction is that it is a mechanical process – one that doesn't require any creative thought. In other words, it can be automated. A computer program can be written to perform this conversion.

If two FA's each have a single start state and a single final state, and if the start state doesn't have any inbound edges and the final state doesn't have any outbound edges, then the two FA's can be composed by connecting the end state of one FA to the start state of the other using a Λ transition. Thompson's Construction makes use of this fact by creating composed FA's for each operation supported by regular expressions (concatenation, alternation, and Kleene Closure). An NFA can be built for any regular expression simply by composing it one small piece at a time using Thompson's three diagrams. Figure 5.3 shows the three base diagrams.

The trick to using Thompson's construction is to NOT be creative. Each FA built with Thompson's has a single start and a single final state. The diagram can be dropped as-is directly into the next step in the construction. Nodes never need to be erased, and each composition should be drawn exactly as shown in Figure 5.3.

Let's illustrate by doing several constructions. First, let's construct an NFA for `a (b | c)`. It's best to start with the inner most operation (in this case `(b | c)`) and work out. So first draw the alternation diagram, and then drop

[1]There are proofs of this statement, but they are normally left to a course on computer theory.

[2]Credited to Ken Thompson, the originator of the Unix operating system.

the resulting diagram into the concatenation diagram. This is illustrated in Figure 5.4.

zzz

5.3 Subset Construction

This section presents the Subset Construction

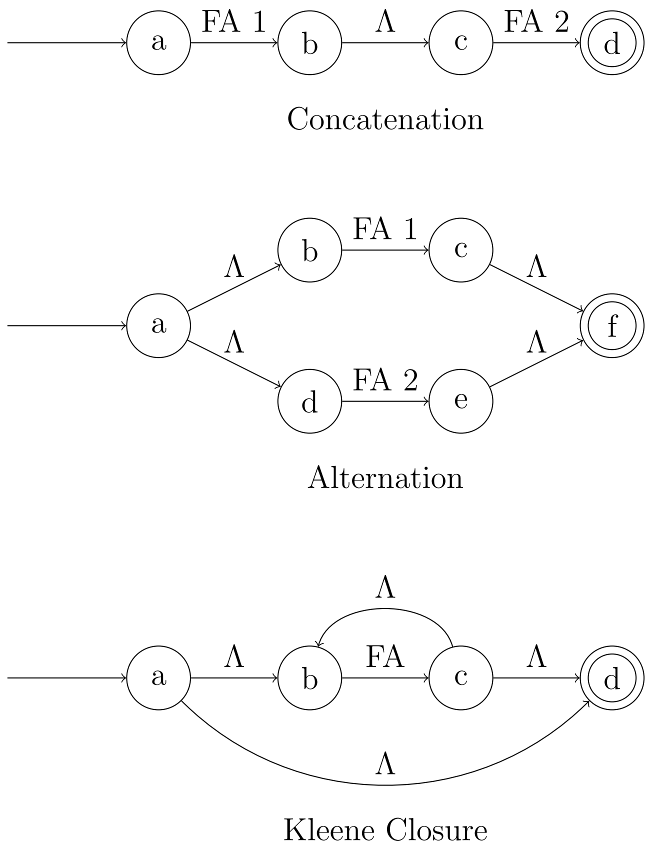
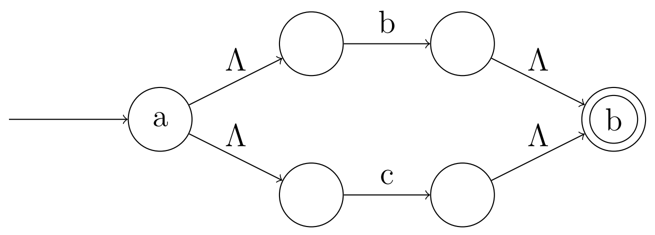
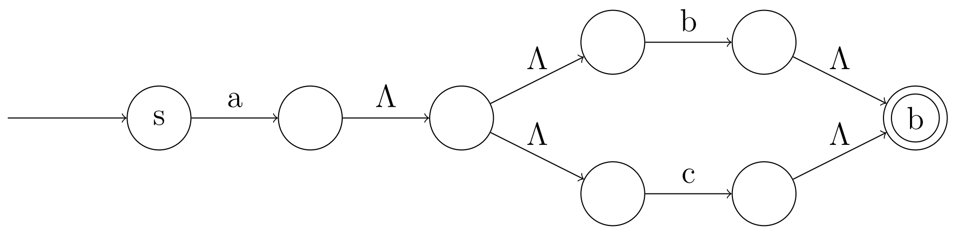


Figure 5.3: Thompson's Construction method makes use of these three diagrams. For each diagram, the FA(s) being composed are represented as two nodes (the start and end nodes) labeled FA, $FA1$, or $FA2$.



`(b | c)`



`a (b | c)`

Figure 5.4: Thompson's Construction for `a (b | c)`. Note that in the second step, the Nodes *a* and *b* from the first step are dropped into the position of Nodes *c* and *d* for the concatenation 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

25

Chapter 9

Top-down recursive-descent parsers

Chapter 10

Bottom-up parsers

Chapter 11

Automatically generated parsers

Description of bison.

Part IV

Semantic Processing

Chapter 12

Syntax vs. Semantics

Chapter 13

Type systems

Chapter 14

Implementation details

Part V

The Back-end

Chapter 15

The Visitor Pattern

Chapter 16

Code generation

Chapter 17

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.

17.1 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.