Preface

Many compiler books have been published over the years, so why another one? Let me be perfectly clear. This book is designed primarily for use as a textbook in a one-semester course for undergraduate students and beginning graduate students. The only prerequisites for this book are familiarity with basic algorithms and data structures (lists, maps, recursion, etc.), a rudimentary knowledge of computer architecture and assembly language, and some experience with the Java programming language or a closely related language such as Kotlin or C#. (A separate edition of this book is available that uses Kotlin as the implementation language for the compiler.) Most undergraduate computer science majors will have covered these topics in their first two years of study. Graduate students who have never had a course in compilers will also find the book useful, especially if they undertake some of the more challenging project exercises described in Appendix B.

A complete study of compilers could easily fill several graduate-level courses, and therefore some simplifications and compromises are necessary for a one-semester course that is accessible to undergraduate students. Following are some of the decisions made in order to accommodate the goals of this book.

1. The book has a narrow focus as a project-oriented course on compilers. Compiler theory is kept to a minimum, but the project orientation retains the "fun" part of studying compilers.
2. The source language being compiled is relatively simple, but it is powerful enough to be interesting and challenging. It has scalar data types, strings, arrays, records, control structures, procedures, functions, and parameters, but it relegates many other interesting language features to the project exercises. Most undergraduate students will find it challenging just to complete a compiler for the basic project language without any additional features. Graduate students will likely want to extend the basic project language with features outlined in the exercises.
3. The target language is assembly language for a virtual machine with a stack-based architecture, similar to but much simpler than the Java Virtual Machine (JVM). This approach greatly simplifies code generation. First, it eliminates the need to deal with general-purpose registers. And second, relative addresses for branch instructions are handled by the assembler, simplifying the amount of work that needs to be done by the compiler. Both an assembler and an emulator for the virtual machine are provided in the book's GitHub repository.
4. No special compiler-related tools are required or used within the book. Students require access only to a Java compiler and a text editor, but most students will want to use Java with an Integrated Development Environment (IDE) such as Eclipse or IntelliJ IDEA. Compiler-related tools such as scanner generators or parser generators could simplify certain tasks involved in building a compiler, but I believe that the approach used in this book makes the structure of the compiler more transparent. Students who wish to use compiler-related tools are welcome to do so, but they will need to look elsewhere to learn how to use these tools. Examples of freely available compiler-related tools include ANTLR, Coco/R, Flex/Bison, Lex/Yacc, JavaCC, Truffle,

and Eclipse Xtext. In addition, while the presentation of the book uses Java, students are free to select an alternative implementation language. Languages that support recursion and object-oriented programming will work best with the approach used in this book. Examples include Kotlin, C#, C++, Python, Scala, and Swift.

5. One very important component of a compiler is the parser, which verifies that a source program conforms to the language syntax and produces an intermediate representation of the program that is suitable for additional analysis and code generation. There are several different approaches to parsing, but in keeping with the emphasis on a one-semester course, this book covers only one approach – recursive descent parsing with several lookahead tokens. A previous edition of this book used only one lookahead token, but changes to the source language motivated the need for additional lookahead tokens. Plus, it is instructive to learn how to implement a parser using multiple lookahead tokens, especially since multiple lookahead tokens can be implemented simply and efficiently.
6. Missing from the book are a lot of favorite compiler topics such as finite automata, bottom-up parsing, attribute grammars, heap management, register allocation, and data-flow analysis. What remains fits nicely into a one-semester, project-oriented course for undergraduate students and beginning graduate students.

Why Study Compilers?

Relatively few people write commercial compilers, but I believe that the study of compiler design is an important part of a computer science education since compilers and closely related language translators are the primary tools for software development. Having a fundamental knowledge of how compilers actually work can improve one's ability to write good programs. In addition, learning how to write a compiler gives experience on working with a moderately large program, and it makes an excellent case study for software engineering and good program design. And the techniques used in compiler design are useful for other purposes such as writing a test harness, a support tool for process control, or an interpreter for a small, special-purpose language. One of my former students applied compiler technology to a commercial tool for testing software.

The Course Project

This book discusses implementation of a compiler for a relatively small programming language named CPRL (for Compiler PRoject Language), which was designed for teaching basics of compiler construction. The target language is assembly language for CVM (CPRL Virtual Machine), a simple stack-based virtual machine. Together we will build a compiler slowly, one step at a time, with lots of template Java code in the book and even more in the book's GitHub repository to guide you through the process. Students are expected to download and study the Java code from the book repository as part of their learning experience. The end result is likely the largest single program that most undergraduate students will encounter.

Appendix A describes the decomposition of the overall project of developing a CPRL compiler into 11 smaller subprojects (numbered 0-10). Students should complete the subprojects in the specified order since each one builds on the previous.

Also, we will build the parser in three separate subprojects, focusing initially on the syntax as defined by the context-free grammar, then adding error recovery, and finally constructing abstract syntax trees for subsequent phases of constraint analysis and initial code generation. In the real world we would likely combine all three steps into one, but the division into three separate subprojects has been shown to provide a better understanding of the concepts and overall process for students new to compiler design.

Appendix B describes a number of extensions or variations to the basic compiler project outlined in this book. Some are more challenging than others. Ambitious undergraduate students and most graduate students will want to attempt one or more of these exercises.

Changes in the Fourth Edition

The fourth edition retains the objectives, emphasis, and general structure of the first three editions, but there are several changes. One simplification to the compiler project is that Java modules are no longer used; i.e., the project is structured using only packages and `CLASSPATH`. In addition, package `edu.citadel.compiler` has been renamed to `edu.citadel.common` to better reflect its role in the project. Other changes can be grouped into two broad categories.

1. **CPRL Language Changes.**

   - A `forLoop` statement has been incorporated directly into the definition of CPRL rather than leaving its addition as an exercise. Technically, a `forLoop` was not strictly necessary since CPRL already incorporated a `while` loop, but a `forLoop` greatly simplifies many programming situations, especially when working with arrays. Several of the test examples were rewritten to take advantage of the new `forLoop`. The syntax of the new `forLoop` was motivated by the Ada programming language and is illustrated in the following example.

     ```
     for i in 1..10 loop
         writeln a[i];
     ```

     The loop variable (`i` in the above example) is implicitly declared to have type `Integer` and is scoped to the body of the loop statement. A new reserved word "`in`" and a new symbol "`..`" (dot dot) were added to the language in support of the `forLoop`. Note that "`for`" was already an unused CPRL reserved word in anticipation of adding a `forLoop` at some time in the future.

   - Array and string variables can now be created without first creating a new type, as illustrated below.

     ```
     var a : array[20] of Integer;
     var s : string[50];
     ```

However, type names are required when passing arrays and strings as parameters to subprograms. Type equivalence is now defined slightly differently for arrays and strings to accommodate this new feature.

- New bitwise operators and shift operators were added for integers.
  - `&` (bitwise and)
  - `|` (bitwise or)
  - `^` (bitwise exclusive or)
  - `~` (bitwise not; a.k.a. bitwise complement)
  - `<<` (left shift)
  - `>>` (signed right shift)
- Arrays and records can now be initialized when they are declared, and the initializers can be nested; e.g., for arrays of arrays or arrays of records. Composite initializer values are enclosed in braces and separated by commas. Here are some examples.

```
var a : array[10] of Integer := { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

type Point = record
               {
                 x : Integer;
                 y : Integer;
               };
var pa : array[4] of Point := {
                                {  1,  0 },
                                {  0,  1 },
                                { -1,  0 },
                                { -1, -1 }
                              };
```

- Additionally, there were several other relatively minor changes to CPRL for this edition.
  - Numeric literal values can now be expressed using hexadecimal and binary notation.

```
var x1 : Integer := 0x00FF0000;
var x2 : Integer := 0b00111111;
```

  - Values for numeric constants can now be negative integers. Previously they were restricted to only nonnegative values.
  - Class `AbstractToken` has been removed since it provided very little behavior for inheritance.
  - Class `TokenBuffer` has been renamed `BoundedBuffer`, reimplemented using generics, and moved to the `edu.citadel.common` package. For the CPRL compiler, the buffer is instantiated with type `Token`. Previously type `Token` was hardcoded into the buffer.

2. CVM Changes.

- Several new or previously unused instruction opcodes are now used to support shift and bit-level operators.

  ```
  BITAND   BITOR   BITXOR   BITNOT   SHL   SHR
  ```

  Technically, the shift opcodes appeared since the first edition of this book to support code optimization, but they have been rewritten to make them similar to the way they work on the JVM, and they are no longer used for optimization.

- Some of the operand byte codes changed from earlier editions, so source code might need to be recompiled and reassembled in order to work properly with the current version of CVM.

In addition, there are many new paragraphs, various rewordings to make the presentation easier to understand, and a few corrections. In developing this new edition, several other possible changes to CPRL were considered, implemented, and then rejected to keep the book at a reasonable length and level of complexity. Overall, I am extremely pleased with the resulting new edition of this book.

Structure of the Book

The book chapters are organized in a natural progression from general concepts to step-by-step details for implementing the compiler project. Each chapter builds on the previous chapters, and forward references have been kept to a minimum. A few sections can be omitted or postponed without loss of continuity, but in general, the book is designed to be studied from front to back.

Chapter 1 introduces basic concepts and terminology in the study of compilers. Following the book by David A. Watt and Deryck F. Brown [Watt 2000], Chapter 1 uses tombstone diagrams as a visual aid to explain many of the concepts. Sections 1.2, 1.3, and 1.6 are required. The other sections can be omitted or postponed without impacting the material in subsequent chapters, but doing so is not recommended.

Chapter 2 presents an overview of the structure of a compiler. In a sense, it sets the context for the remaining chapters.

Chapter 3 provides an overview of how programming languages are defined, with detailed coverage of context-free grammars. The complete context-free grammar for CPRL, the source language for the compiler project, is defined in Appendix D.

Chapter 4 gives a brief overview of CPRL. A more complete definition of CPRL is provided in Appendices C and D.

Chapter 5 presents the details of implementing a scanner for CPRL, including an overview of several related classes found in the book's GitHub repository.

Chapter 6 is the longest and one of the most complicated chapters in the book. It includes a discussion of grammar analysis to compute first sets and follow sets plus details on the implementation of a recursive descent parser for CPRL. As defined in the compiler project,

the initial version of the parser concentrates primarily on verification that a source program conforms to the syntax defined by the context-free grammar for CPRL.

Chapter 7 extends the parser developed in Chapter 6 to perform error recovery so that multiple errors can be detected and reported by the compiler.

Chapter 8 contains a detailed discussion of abstract syntax trees. Abstract syntax trees provide an intermediate representation for source programs that can be used for additional analysis and code generation. In this chapter, the parsing methods are modified to construct an abstract syntax tree for the source program.

Chapter 9 uses the abstract syntax trees from Chapter 8 to perform constraint analysis, with emphasis on checking conformance to CPRL's type rules as defined for CPRL/`0`, a major subset of CPRL.

Chapter 10 gives a brief overview of the CVM, the virtual machine that serves as the target for CPRL programs. The CVM has a stack architecture that is similar to but much simpler than the Java Virtual Machine (JVM). Appendix E contains additional information about the CVM including complete definitions for every CVM instruction.

Chapter 11 modifies the abstract syntax trees to generate assembly language for CPRL/`0`. The GitHub repository provides an assembler that can be used to generate the actual CVM machine code.

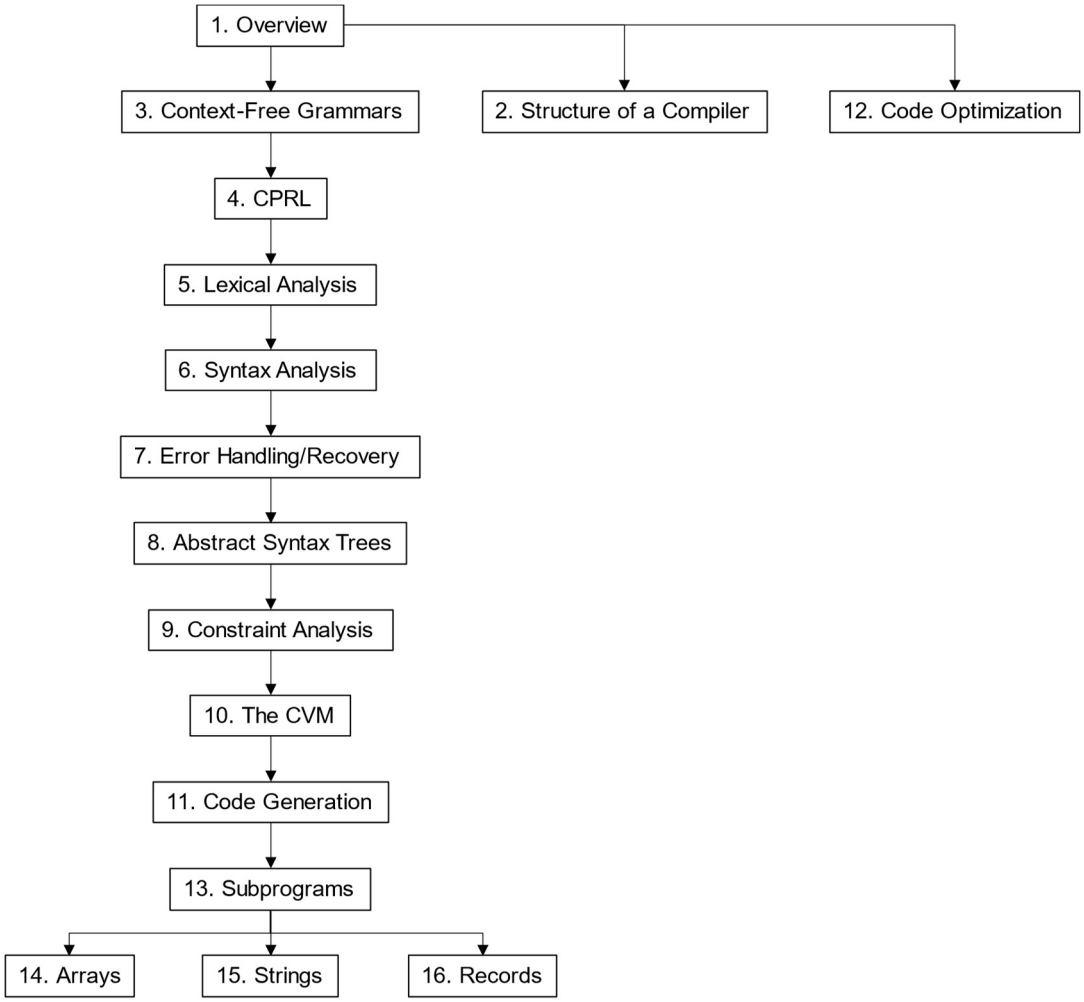
Chapter 12 presents an overview of code optimization. The assembler provided on the course web site implements several optimizations, but none of the compiler projects require mastery of the material from this chapter. Therefore, this chapter can be postponed if desired, but it should not be omitted.

Chapter 13 extends constraint analysis and code generation to handle subprograms. It includes additional coverage of scope and a detailed discussion of calling conventions and activation records. Similar to Chapter 6, this chapter is long and can be quite complicated for many undergraduate students. But mastery of this material is essential for a course on compilers, even if it means that less time is available for the last three chapters.

Chapters 14, 15, and 16 extend constraint analysis and code generation for arrays, strings, and records, respectively. Implementing records completes the basic compiler project, but students are always encouraged to attempt some of variations and extensions outlined in Appendix B.

Chapter Dependencies

As illustrated in the following diagram, Chapters 2 and 12 can be covered any time after Chapter 1, and Chapters 14-16 can be covered any time after Chapter 13. Otherwise, most chapters should be covered in the order presented in the book. The recommended approach is to cover the chapter in order 1-16 using the appendices as supplementary material.



Book Resources

The repository for this book at https://github.com/SoftMoore/CPRL-Java-4th contains a number of related resources as follows.

- Java source code that implements the CVM, the target machine for the compiler project.
- Java source code that implements an assembler for the CVM. The compiler project targets assembly language for the CVM rather than actual virtual machine bytecode.
- Java source code for a disassembler; i.e., a program that takes CVM machine code and converts it back into assembly language. This program can be useful for debugging the compiler projects and for understanding how machine code is laid out in memory.

- Java source code or skeletal Java source code for many of the classes described in the book so that students don't need to start from scratch to create their compilers. Much of the code for these classes reveal implementation ideas for other classes whose implementations are either not provided or are only partially provided. Students should begin each phase of their compiler project by trying to understand the related Java source code that is provided.
- Many examples of correct and incorrect CPRL programs that can be used to test a compiler. The final compiler should reject all incorrect CPRL programs with appropriate error messages, and it should generate semantically equivalent assembly language for all correct CPRL programs. Using the provided assembler and CVM emulator, all correct CPRL programs can be run to compare observed and expected results. Students are strongly encouraged to develop additional test programs.
- Sample Windows command scripts and Bash shell scripts for running and testing various stages of the compiler. For example, rather than trying to compile and test programs one at a time, there are command/shell scripts for running the compiler on all CPRL source files in the current working directory. These scripts are useful for testing the compiler against collections of correct and incorrect CPRL programs.

Appendix A provides additional details about the resources available on the book repository and how those resources fit into the overall structure of the course compiler project.

Java and Object-Orientation

As indicated in the book's title, the language used herein for implementing a compiler is Java, but other options are certainly possible. In fact, one of the project exercises listed in Appendix B is to write the compiler in a different language, and some of my former students have done this.

Since Java is an object-oriented language, then every component of the compiler is structured in terms of classes and objects – the scanner is an object, the parser is an object, tokens are objects, etc. But the full power of "object-orientation" doesn't come into play until Chapter 8, where the details of abstract syntax trees are introduced. The approach used in this book for implementing abstract syntax trees fully exploits Java's object-oriented facilities for inheritance and polymorphism. As an example, consider that code generation for a list of statements looks like the following.

```
for (Statement stmt : statements)
    stmt.emit();
```

The list could include `if` statements, assignment statements, loop statements, etc., and therefore different statements generate code in different ways. But at the level of programming abstraction shown in this example, we only need to know that each statement has an `emit()` method that writes out the appropriate object code.

Similarly, the code to check constraints for a list of CPRL subprogram declarations looks like the following.

```
for (SubprogramDecl decl : subprogDecls)
    decl.checkConstraints();
```

Note that the subprogram declarations in the list could be either procedures or functions.

Which Version of Java?

The exposition in this book and the resources provided in the book's GitHub repository assume Java version 17 or later. In particular, interfaces `VariableDecl` and `Initializer` are declared to be sealed interfaces, and several classes make use of pattern matching for `instanceof`.

The Java source code also makes use of the newer arrow syntax for switch statements and switch expressions, and Java source files frequently use local variable type inference with the keyword `var`, especially in conjunction with the `new` operator. Here is an example.

```
ArrayList<Statement> statements = new ArrayList<>(20);   // older style
var statements = new ArrayList<Statement>(20);           // using var
```

A Note from the Author About Formatting Source Code

Every programmer has his/her preferred way to format and indent source code. Should curly braces be aligned? Should code be indented 2, 3, or 4 spaces? What is the best way to format an `if` statement or a `while` loop? Sometimes arguments about how source code should be formatted seem to take on religious-like fervor. I avoid such arguments, for I have found the **one true way** that Java source code should be formatted. ☺ But if you are a nonbeliever, please feel free to reformat the Java source code according to your project standards or personal tastes. (FYI: The formatting style used in this book is loosely based on the GNU style, which is not very popular but which I have personally found to be the most readable. As the saying goes, "Beauty is in the eye of the beholder.")

Acknowledgements

When I was first learning about computers as a young man, I was in awe of compilers. That awe evolved into curiosity, which led to a lot of thought and study about how compilers are constructed, which led to a lot of contemplation about how best to teach an introductory course in compilers, which eventually led to my writing this book. Along the way I was guided and influenced by a number of teachers, colleagues, students, and books. I would like to acknowledge those that have been the most influential on my thinking.

I was first introduced to compilers many years ago when I "unofficially" audited a course on compiler design given by Richard LeBlanc at Georgia Institute of Technology (a.k.a. Georgia Tech). For reasons I can no longer remember, I was unable to take the course for credit, but auditing it, especially under Richard LeBlanc, was enough to motivate me to learn more. I was fortunate enough to have Richard LeBlanc later on for another course on programming language design.

My next introduction to compilers came in a professional development course given by Frank DeRemer and Tom Pennello, with a guest lecture or two by Bill McKeeman. They will not remember me, but I am grateful to have learned more about compilers from teachers and researchers of their calibers.

I have also been inspired by several compiler books, especially two of them that took a pedagogical approach similar to this one. The first book is *Brinch Hansen on Pascal Compilers* by Per Brinch Hansen (Prentice Hall, 1985). That book is a little out of date now, but it had one of the most readable descriptions of compilers when it was first released. A second, much more modern book is *Programming Language Processors in Java: Compilers and Interpreters* by David A. Watt and Deryck F. Brown (Prentice Hall 2000). I followed their treatment of tombstone diagrams when explaining compilers and interpreters in Chapter 1. Years ago I used the Brinch Hansen book as a textbook in my compiler courses, and I used the Watt-Brown book several times when it was first published.

It is important to acknowledge former students at Johns Hopkins University and The Citadel, and to thank these institutions for allowing me to explore my ideas about writing compilers. I am particularly grateful to the following students who had the most influence on my thinking about how to teach compilers: Rob Ring, Scott Stanchfield, Zack Aardahl, Ben Hunter, Dalton Hazelwood, Gordon Finlay, Josh Terry, Davis Jeffords, Mike Dalpee, Matthew Blair, Mafer Contreras, Jared Johnson, Robert Roser, Shiloh Smiles, Noah Klepper, and Robert Powell . Some of the earlier students at Johns Hopkins suffered through my courses as I was trying to crystallize my approach to teaching compilers, and I am grateful for their feedback. They would barely recognize my course if they took it today.

I must also acknowledge Vince Sigillito, who served as Chair of the Computer Science program at the Johns Hopkins Part-Time Programs in Engineering and Applied Science, for first allowing me to teach a course on compilers many years ago. I remember telling Vince that I wanted to teach a course on compilers so that I could learn more about them. He had no objections and even indicated that he had done something similar in the past. Compiler Design has evolved into my favorite course.

I would especially like to acknowledge the following individuals who provided invaluable advice and feedback on the presentation and exposition in this book: Richard LeBlanc, Art Pyster, Shankar Banik, and George Rudolph. I am most appreciative and humble that they spent part of their valuable time assisting me with this effort.

Finally, I would like to acknowledge Kayran Cox Moore, my wife of many years, for proofreading and providing invaluable feedback on several drafts of this book. She might not understand compilers or Java programming, but she has a deep understanding of English grammar and sentence structure, and she has no reservations about correcting my errors or improving my writing. Any grammatical errors remaining in this book are a result of my stubborn refusal to follow her advice, or, more likely, they were introduced after she proofread that part of the book. I also want to thank Kayran for being my anchor in life and the source for most of what is good about myself.