Preface

Why Another Compiler Text?

There are lots of compiler texts out there. Some of them are very good. Some of them use Java as the programming language in which the compiler is written. But we have yet to find a compiler text that uses Java everywhere.

Our text is based on examples that make full use of Java:

- Like some other texts, the implementation language is Java. And, our implementation uses Java's object orientation. For example, polymorphism is used in implementing the `analyze()` and `codegen()` methods for different types of nodes in the abstract syntax tree (AST). The lexical analyzer (the token scanner), the parser, and a back-end code emitter are objects.

- Unlike other texts, the example compiler and examples in the chapters are all about compiling Java. Java is the source language. The student gets a compiler for a non-trivial subset of Java, called *j--*; *j--* includes classes, objects, methods, a few simple types, a few control constructs, and a few operators. The examples in the text are taken from this compiler. The exercises in the text generally involve implementing Java language constructs that are not already in *j--*. And, because Java is an object-oriented language, students see how modern object-oriented constructs are compiled.

- The example compiler and exercises done by the student target the Java Virtual Machine (JVM).

- There is a separate back end (discussed in Chapters 6 and 7), which translates a small but useful subset of JVM code to SPIM (Larus, 2000–2010), a simulator for the MIPS RISC architecture. Again, there are exercises for the student so that he or she may become acquainted with a register machine and register allocation.

The student is immersed in Java and the JVM, and gets a deeper understanding of the Java programming language and its implementation.

Why Java?

It is true that most industrial compilers (and many compilers for textbooks) are written in either C or C++, but students have probably been taught to program using Java. And few students will go on to write compilers professionally. So, compiler projects steeped in Java give students experience working with larger, non-trivial Java programs, making them better Java programmers.

A colleague, Bruce Knobe, says that the compilers course is really a software engineering course because the compiler is the first non-trivial program the student sees. In addition, it is a program built up from a sequence of components, where the later components depend on the earlier ones. One learns good software engineering skills in writing a compiler.

Our example compiler and the exercises that have the student extend it follow this model:

- The example compiler for *j--* is a non-trivial program comprising 240 classes and nearly 30,000 lines of code (including comments). The text takes its examples from this compiler and encourages the student to read the code. We have always thought that reading good code makes for better programmers.
- The code tree includes an Ant file for automatically building the compiler.
- The code tree makes use of JUnit for automatically running each build against a set of tests. The exercises encourage the student to write additional tests before implementing new language features in their compilers. Thus, students get a taste of extreme programming; implementing a new programming language construct in the compiler involves
  - Writing tests
  - Refactoring (re-organizing) the code for making the addition cleaner
  - Writing the new code to implement the new construct

The code tree may be used either

- In a simple command-line environment using any text editor, Java compiler, and Java run-time environment (for example, Oracle's Java SE). Ant will build a code tree under either Unix (including Apple's Mac OS X) or a Windows system; likewise, JUnit will work with either system; or
- It can be imported into an integrated development environment such as IBM's freely available Eclipse.

So, this experience makes the student a better programmer. Instead of having to learn a new programming language, the student can concentrate on the more important things: design, organization, and testing. Students get more excited about compiling Java than compiling some toy language.

Why Start with a *j--* Compiler?

In teaching compiler classes, we have assigned programming exercises both

1. Where the student writes the compiler components from scratch, and
2. Where the student starts with the compiler for a base language such as *j--* and implements language extensions.

We have settled on the second approach for the following reasons:

- The example compiler illustrates, in a concrete manner, the implementation techniques discussed in the text and presented in the lectures.
- Students get hands-on experience implementing extensions to *j--* (for example, interfaces, additional control statements, exception handling, doubles, floats and longs, and nested classes) without having to build the infrastructure from scratch.
- Our own work experiences make it clear that this is the way work is done in commercial projects; programmers rarely write code from scratch but work from existing code bases. Following the approach adopted here, the student learns how to fit code into existing projects and still do valuable work.

Students have the satisfaction of doing interesting programming, experiencing what coding is like in the commercial world, and learning about compilers.

Why Target the JVM?

In the first instance, our example compiler and student exercises target the Java Virtual Machine (JVM); we have chosen the JVM as a target for several reasons:

- The original Oracle Java compiler that is used by most students today targets the JVM. Students understand this regimen.
- This is the way many compiler frameworks are implemented today. For example, Microsoft's .NET framework targets the Common Language Runtime (CLR). The byte code of both the JVM and the CLR is (in various instances) then translated to native machine code, which is real register-based computer code.
- Targeting the JVM exposes students to some code generation issues (instruction selection) but not all, for example, not register allocation.
- We think we cannot ask for too much more from students in a one-semester course (but more on this below). Rather than have the students compile toy languages to real hardware, we have them compile a hefty subset of Java (roughly Java version 4) to JVM byte code.
- That students produce real JVM .class files, which can link to any other .class files (no matter how they are produced), gives the students great satisfaction. The class emitter (`CLEmitter`) component of our compiler hides the complexity of .class files.

This having been said, many students (and their professors) will want to deal with register-based machines. For this reason, we also demonstrate how JVM code can be translated to a register machine, specifically the MIPS architecture.

After the JVM – A Register Target

Beginning in Chapter 6, our text discusses translating the stack-based (and so, register-free) JVM code to a MIPS, register-based architecture. Our example translator does only a

limited subset of the JVM, dealing with static classes and methods and sufficient for translating a computation of factorial. But our translation fully illustrates linear-scan register allocation—appropriate to modern just-in-time compilation. The translation of additional portions of the JVM and other register allocation schemes, for example, that are based on graph coloring, are left to the student as exercises. Our JVM-to-MIPS translator framework also supports several common code optimizations.

Otherwise, a Traditional Compiler Text

Otherwise, this is a pretty traditional compiler text. It covers all of the issues one expects in any compiler text: lexical analysis, parsing, abstract syntax trees, semantic analysis, code generation, limited optimization, register allocation, as well as a discussion of some recent strategies such as just-in-time compiling and hotspot compiling and an overview of some well-known compilers (Oracle's Java compiler, GCC, the IBM Eclipse compiler for Java and Microsoft's C# compiler). A seasoned compiler instructor will be comfortable with all of the topics covered in the text. On the other hand, one need not cover everything in the class; for example, the instructor may choose to leave out certain parsing strategies, leave out the JavaCC tool (for automatically generating a scanner and parser), or use JavaCC alone.

Who Is This Book for?

This text is aimed at upper-division undergraduates or first-year graduate students in a compiler course. For two-semester compiler courses, where the first semester covers front-end issues and the second covers back-end issues such as optimization, our book would be best for the first semester. For the second semester, one would be better off using a specialized text such as Robert Morgan's *Building an Optimizing Compiler* [Morgan, 1998]; Allen and Kennedy's *Optimizing Compilers for Modern Architectures* [Allen and Kennedy, 2002]; or Muchnick's *Advanced Compiler Design and Implementation* [Muchnick, 1997]. A general compilers text that addresses many back-end issues is Appel's *Modern Compiler Implementation in Java* [Appel, 2002]. We choose to consult only published papers in the second-semester course.

Structure of the Text

Briefly, *An Introduction to Compiler Construction in a Java World* is organized as follows. In Chapter 1 we describe what compilers are and how they are organized, and we give an overview of the example *j--* compiler, which is written in Java and supplied with the text. We discuss (lexical) scanners in Chapter 2, parsing in Chapter 3, semantic analysis in Chapter 4, and JVM code generation in Chapter 5. In Chapter 6 we describe a JVM code-to-MIPS code translator, with some optimization techniques; specifically, we target

James Larus's SPIM, an interpreter for MIPS assembly language. We introduce register allocation in Chapter 7. In Chapter 8 we discuss several celebrity (that is, well-known) compilers. Most chapters close with a set of exercises; these are generally a mix of written exercises and programming projects.

There are five appendices. Appendix A explains how to set up an environment, either a simple command-line environment or an Eclipse environment, for working with the example *j--* compiler. Appendix B outlines the *j--* language syntax, and Appendix C outlines (the fuller) Java language syntax. Appendix D describes the JVM, its instruction set, and `CLEmitter`, a class that can be used for emitting JVM code. Appendix E describes SPIM, a simulator for MIPS assembly code, which was implemented by James Larus.

How to Use This Text in a Class

Depending on the time available, there are many paths one may follow through this text. Here are two:

- We have taught compilers, concentrating on front-end issues, and simply targeting the JVM interpreter:
  - Introduction. (Chapter 1)
  - Both a hand-written and JavaCC generated lexical analyzer. The theory of generating lexical analyzers from regular expressions; Finite State Automata (FSA). (Chapter 2)
  - Context-free languages and context-free grammars. Top-down parsing using recursive descent and LL(1) parsers. Bottom-up parsing with LR(1) and LALR(1) parser. Using JavaCC to generate a parser. (Chapter 3)
  - Type checking. (Chapter 4)
  - JVM code generation. (Chapter 5)
  - A brief introduction to translating JVM code to SPIM code and optimization. (Chapter 6)
- We have also taught compilers, spending less time on the front end, and generating code both for the JVM and for SPIM, a simulator for a register-based RISC machine:
  - Introduction. (Chapter 1)
  - A hand-written lexical analyzer. (Students have often seen regular expressions and FSA in earlier courses.) (Sections 2.1 and 2.2)
  - Parsing by recursive descent. (Sections 3.1 3.3.1)
  - Type checking. (Chapter 4)
  - JVM code generation. (Chapter 5)
  - Translating JVM code to SPIM code and optimization. (Chapter 6)
  - Register allocation. (Chapter 7)

In either case, the student should do the appropriate programming exercises. Those exercises that are not otherwise marked are relatively straightforward; we assign several of these in each programming set.