An Introduction to Compiler Construction in a Java World

Bill Campbell, Swami Iyer and Bahar Akbal-Delibaş

The University of Massachusetts Boston

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 makes full use of *Java*.

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

- Like some other texts, the implementation language is *Java*. And, the implementation makes full use of *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, in the first instance targets the Java Virtual Machine (JVM).

- There is a separate back end, 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 she may become acquainted with a register machine and register allocation.

Thus, 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.

An old colleague, Bruce Knobe used to say that the compilers course is really a software engineering course because the compiler is the first non-trivial program that 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 runtime 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, she can concentrate on the more important things: design, organization and testing. Students get more excited about compiling *Java* than compiling some toy language.

Why Target the JVM?

In the first instance, our example compiler and student exercises target the Java Virtual Machine; 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 regime.

- 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, e.g. 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 a 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 7, 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, e.g. that 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, 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 cover 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* (Digital Press, 1998); Allen and Kennedy, *Optimizing Compilers for Modern Architectures* (Morgan Kaufmann, 2001); or Muchnick's *Advanced Compiler Design and Implementation* (Morgan Kaufmann, 1997). A general compilers text that addresses many back-end issues is Appel's with *Modern Compiler Implementation in Java, Second Edition* (Cambridge University Press, 2004).

The 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. We describe strategies for implementing new *Java* constructs that are not already in the *j--* subset in Chapter 6. In Chapter 7 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 8. In Chapter 9 we discuss a few 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, it's instruction set, and CLEmitter, a class that can be used for emitting JVM code. Appendix E briefly describes the MIPS architecture and 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 handwritten 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)
  - Extensions to the *j--* compiler. (Sections in 6.2 as appropriate)

- 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 handwritten 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)
  - Extensions to the *j--* compiler. (Sections in 6.2 as appropriate)
  - Translating JVM code to SPIM code. (Chapter 7)
  - Register Allocation. (Chapter 8)

In either case, the student should do the appropriate programming exercises.

Where to Get the Code

We supply a code tree, containing

- *Java* code for the example *j--* compiler and the JVM to SPIM translator,
- tests (both conformance tests and deviance tests that cause error messages to be produced) for the *j--* compiler and a framework for adding additional tests,
- the JavaCC and Junit libraries, and
- an *Ant* file for building and testing the compiler.

We maintain a web site at http://www.cs.umb.edu/j-- for up-to-date distributions.

What Does the Student Need?

The code tree may be obtained at http://www.cs.umb.edu/j--/j--.zip.

Everything else the student needs is freely obtainable on the WWW: the latest version of *Java SE* is obtainable from Oracle at http://www.oracle.com/technetwork/java/javase/downloads/index.html. *Ant* is available at http://ant.apache.org/; and *Eclipse* can be obtained from http://www.eclipse.org/. All of this may be installed on *Windows*, *Mac OS X*, or any *Linux* platform.

What Does the Student Come Away With?

The student gets hands-on experience working with and extending (in the exercises) a real, working compiler. From this, the student gets an appreciation of how compilers work, how to write compilers, and how the *Java* language behaves. More importantly, the student gets practice working with a non-trivial *Java* program of more than 25,000 lines of code.

The Authors

Bill Campbell is an associate professor in the Department of Computer Science at the University of Massachusetts, Boston. His professional areas of expertise are software engineering, object-oriented analysis, design and programming, and programming language implementation. He likes to write programs. He has both academic and commercial experience. He has been teaching compilers for more than twenty years. He has written an introductory *Java* programming text with Ethan Bolker, *Java Outside In* (Cambridge University Press, 2003).

He has worked for (what is now) AT&T and Intermetrics Inc., and has consulted to Apple Computer and Entitlenet. He has implemented a public domain version of the *Scheme* programming language called *UMB Scheme*, which is distributed with Linux. Recently, he founded an undergraduate program in information technology.

He has a bachelor's degree in mathematics and computer science from New York University, 1972; an M.Sc. in computer science from McGill University, 1975; and a PhD in computer science from St Andrews University (UK), 1978.

Swami Iyer is a PhD candidate in the Department of Computer Science at the University of Massachusetts, Boston. His research interests are in the fields of dynamical systems, complex networks, and evolutionary game theory. He also has a casual interest in theoretical physics. His fondness for programming is what got him interested in compilers. He has been working on the *j--* compiler for several years.

He enjoys teaching and has taught classes in introductory programming and data structures at the University of Massachusetts, Boston. After graduation, he plans on pursuing an academic career with both teaching and research responsibilities.

He has a bachelor's degree in electronics and telecommunication from the University of Bombay (India), 1996 and a master's degree in computer science from the University of Massachusetts, Boston, 2001.

Bahar Akbal-Delibaş is a PhD student in the Department of Computer Science at the University of Massachusetts, Boston. Her research interest is in Structural Bioinformatics, aimed in better understanding the sequence-structure-function relationship in proteins, modeling conformational changes in proteins and predicting protein-protein interactions. She also performed research on Software Modeling, specifically modeling wireless sensor networks. She enjoys programming, especially when programs run without any compiler errors.

Her first encounter with Compilers was a frightening experience as it can be for many students. However, soon she discovered how to play with the pieces of the puzzle and saw the fun in programming compilers. She hopes this book will help students who read it the same way. She has been the teaching assistant for the Compilers course at the University of Massachusetts, Boston and working with the j-- compiler for several years

She has a bachelor's degree in Computer Engineering from Fatih University (Turkey), 2004; and a master's degree in Computer Science from University of Massachusetts, Boston, 2007.

Acknowledgements

We wish to thank students in CS451 and CS651, the compilers course at the University of Massachusetts Boston for their feedback on, and corrections to, the text, the example compiler and the exercises. We would like to thank Kelechi Dike, Ricardo Menard and Mini Nair for their writing a compiler for a subset of C# that was similar to *j*--. We would particularly like to thank Alex Valtchev for his work on both live intervals and linear scan register allocation.

We wish to thank the people at Taylor & Francis, including Randi Cohen, Jessica Vakili, and the editors for their help in preparing this text.

Finally, we wish to thank our families and close friends for putting up with us as we wrote the compiler and the text.