

Bantam: A Customizable, Java-Based, Classroom Compiler

Marc L. Corliss
Hobart and William Smith Colleges
4170 Scandling Center
Geneva, New York
corliss@hws.edu

E Christopher Lewis
VMware, Inc.
3401 Hillview Ave
Palo Alto, California
lewis@vmware.com

ABSTRACT

This paper introduces the Bantam Java compiler project, a new language and compiler designed specifically for the classroom. Bantam Java, the source programming language, is a small subset of the Java language, which is a commonly-used language in introductory programming courses. Because Bantam Java is similar to Java, it leverages the student's existing intuition and the student can automatically apply what they learn in the course directly to Java. The Bantam Java project is also customizable (it supports several tools and targets), which gives instructors flexibility in designing course assignments. Finally, the Bantam Java compiler project includes a free, comprehensive, student manual, which can be used in conjunction with any compiler textbook.

Categories and Subject Descriptors

D.3.4 [Programming Languages]: Processors—*compilers*

General Terms

Languages, Design

Keywords

Bantam Java

1. INTRODUCTION

Constructing a compiler, the software responsible for translating high-level programs (*e.g.*, Java, C++ programs) into machine code, is a valuable learning experience for an

undergraduate student. Building a compiler naturally requires a deep understanding of the source programming language, the target machine, and everything in between. Very few aspects of computer programming seem like *magic* to a student after successfully implementing a compiler.

A compiler also makes for a challenging software engineering project. In fact, for many programmers it is one of the most challenging applications they will write, so much so that one well-known compiler textbook depicts the compiler as a large dragon on its cover [1]. To deal with this complexity, most compilers make use of modular design, well-organized layers of abstraction, and design patterns such as the visitor pattern [8]. A compiler project gives students experience applying all of these techniques. It also provides students with an opportunity to take part in designing a large software infrastructure.

Another benefit of building a compiler is that it is a wonderful application of many concepts from other areas of computer science including programming languages, computational theory, data structures, algorithms, and complexity analysis. For instance, a compiler relies on finite automaton and context-free grammars (usually covered in a computational theory course) for performing lexical and syntactic analysis. Building a compiler helps students appreciate the importance of these topics via their application in a practical context.

Unfortunately, there are very few compilers that are suitable for an undergraduate course project. Commercial compilers are too complex to be undertaken in a one semester course. At the other extreme, there a number of programming languages and compilers [3, 13], which do not include some of the important features of modern programming languages such as objects and inheritance. Although these languages and compilers abstract aspects of real programming languages, they appear toy-like (and possibly irrelevant) to students. The challenge is building a compiler infrastructure

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGCSE'08 March 12–15, 2008, Portland, Oregon, USA.

Copyright 2008 ACM 978-1-59593-947-0/08/0003 ...\$5.00.

that has most of the important features of modern programming languages but that can be designed and implemented in a one semester course.

In this paper, we present a new programming language and compiler infrastructure that is designed specifically for use in the classroom. It will run on any Linux/x86 machine. The programming language, called Bantam Java (or Bantam for short), is a small subset of the Java language. (Bantam is a city in the north of the Indonesian island of Java and connotes smallness.) Although small, Bantam is sufficiently practical that it can easily be used to write any text-based application. The same cannot be said for many languages compiled by instructional compilers.

Bantam also has several other virtues that make it well-suited for an undergraduate compiler course. First and foremost, the project uses a language similar to Java, which is commonly used in introductory programming courses. This approach leverages the student’s existing intuitions developed in earlier programming courses. The instructor also does not have to argue the relevance of the compiled language; the student instantly believes that the language is relevant and that the leap to a commercial compiler is in their grasp. Moreover, the insights that the student develops in constructing the compiler can be applied directly to Java.

A second virtue of this project is that several aspects of the project are customizable. For instance, this project supports several tools (*e.g.*, lexer and parser generators) and target architectures (*e.g.*, Mips and x86). The components of the project can be easily swapped in and out so that an instructor could, for example, choose to skip lexing and parsing and instead have students implement a garbage collector. The project is also easily extended if an instructor wanted to add an additional language feature such as arrays.

A third virtue of this project is that it includes a free, comprehensive student manual, which can be used in conjunction with any traditional compiler textbook [1, 4, 6, 11]. The manual describes in detail the course assignments, the language’s syntax and semantics, and the overall design of the compiler.

The outline for the remainder of this paper is as follows. The next section presents related work. The subsequent sections discuss the Bantam compiler project in more detail. Section 3 discusses our project goals, which guided both the design of the Bantam language and compiler. Section 4 gives an overview of the Bantam language and discusses the included and excluded Java features in Bantam. Section 5

describes the overall design of the compiler (which must be completed by the student).

2. RELATED WORK

The Bantam project is not the first classroom compiler project, but it does have some advantages over its predecessors. There are a number of compiler projects [3, 13], which do not use object-oriented source languages. The major drawback with these projects is that the dominant programming paradigm today is object-oriented programming. Most of the recent successful commercial languages are object-oriented (*e.g.*, Java, C#, C++, Python).

The COOL (Classroom Object-Oriented Language) project [2] is one example of an object-oriented educational language and compiler. This project uses a source language called COOL, a simple object-oriented language, which the authors formally prove is type safe. While this project is appealing for a compiler course that emphasizes type theory, the language itself is syntactically and semantically different from any other language that students are familiar with. Students must spend time learning the COOL language before beginning the compiler project. Students may also have difficulty applying what they have learned to practical languages like Java.

The MiniJava project [4] is another educational, Java-based compiler project. The MiniJava project is integrated into a particular compiler textbook [4], reducing the project management for the instructor, but this tight integration makes the project an impractical option for instructors wishing to employ a different text. In addition, the object-oriented aspects of the project appear in the “Advanced Topics” section of the text, relegating object-orientation to an optional add-on. Finally, the publisher-provided instructor code is incomplete. The Bantam Java project can be used with any textbook, object-orientation is fundamental, a complete implementation is available to instructors, and it has a complete and clear student manual.

3. PROJECT GOALS

Our design of the Bantam language and compiler was guided by five goals.

Well-known language. Our first goal was to design a language that is a subset of Java, since Java is often used in introductory programming courses. We wanted students to spend less time learning the syntax and semantics of the language and more time focusing on the implementation of

the language. We also wanted students to easily see the relevance of the course project to existing, commercial languages (*i.e.*, Java) and to be able to immediately apply what they learn to Java.

Carefully-selected features. Of course, we could not include all or even most of the Java features in our language. Building a compiler for such a language requires more than a single semester. Our second goal was to select an appropriate subset of Java so that the project retained the character of Java yet resulted in a project that can be realized in a semester-long course.

We chose to emphasize object-oriented programming in our language leaving out such features as static class members, modifiers, interfaces, abstract classes, packages, exceptions, arrays, and most primitives (for a complete list see Section 4). While some of these features add significant intellectual content to the compiler implementation, they all require a significant amount of time to implement.

Well-engineered infrastructure. Our third goal was to build a well-engineered compiler infrastructure. We wanted instructors and students to be able to easily extend our infrastructure. By carefully designing our infrastructure, students can also complete more sophisticated assignments in less time. The compiler project also serves as an example of quality software engineering for students.

Comprehensive documentation. Our fourth goal was to provide a document describing all aspects of the project. To this end we have written a comprehensive student manual, which is freely available off the Bantam Java website (<http://www.bantamjava.com>). This manual will work well in tandem with any traditional compiler textbook [1, 4, 6, 11].

Customizable project. Our fifth and final goal was to give instructors the flexibility to tailor the project to their own classroom needs. For example, instructors can choose between a handful of popular lexer and parser generators as well as target machines (one emulated and one native). Our infrastructure supports using either the JLex scanner generator [5] and the Java Cup parser generator (an LALR parser generator) [9], or the JavaCC scanner and parser generator (an LL(k) parser generator) [12].

The infrastructure also supports two target machines: Mips and x86. The Mips target uses the SPIM emulator [10], which can be downloaded and installed on any x86 machine. The x86 target is a native target. It will work on any 32-

Class	Method signature	Method description
Object	Object clone()	copy an object
Sys	void exit(int status)	exit with status
String	int length()	return string length
	boolean equals(String s)	test if strings are equal
	String substring (int begin, int end)	return substring between begin and end indices
	String concat(String s)	return concatenated string
TextIO	void readfile(String file)	set to read from file
	void writefile(String file)	sets to write to file
	void readStdin()	set to read from stdin
	void writeStdout()	set to write to stdout
	void writeStderr()	set to write to stderr
	String getString()	read next string
	int getInt()	read next int
	void putString(String s)	write string
	void putInt(int i)	write int

Table 1: Bantam built-in classes.

bit, Intel x86 machine running the Linux operating system (although other x86 processors and Unix-based operating systems may work).

In future work, we will explore other targets including the Java Virtual Machine. In addition, we will also allow instructors to choose between building a compiler for the base Bantam language (the one presented in this work) or an extended language. The extended language will include features such as arrays, static class members, and a floating point primitive.

4. BANTAM JAVA LANGUAGE

This section describes at a high level the Bantam Java programming language, which is the source language for the Bantam compiler. As mentioned earlier, we made Bantam Java a subset of the Java language so students are immediately ready to work on the compiler without having to learn a new language. This also gives students experience implementing many real-world language features. The challenge was in choosing the right set of Java features to include in the Bantam language. As illustrated below, we chose to focus on some of the object-oriented features of Java such as objects, inheritance, dynamic dispatch, and object casting.

Types. As Bantam Java is a small language it could not support a lot of types. Bantam has two primitive types and four built-in, object types. The two primitive types are **int** and **boolean**, which are the same as their Java counterparts.

Table 1 shows the Bantam built-in, object types, which include **Object**, **String**, **Sys**, and **TextIO**. **Object** and **String** are taken directly from Java, although the Bantam versions have much less functionality. **Sys** and **TextIO** are not built-

in to Java, however, they easily could be written in Java. **Sys** has an **exit** method for exiting the program, which replaces the static call to **System.exit** in Java. **TextIO** has several methods for performing text-based I/O. **TextIO** supports both reading and writing standard input and output as well as reading and writing files. It was modeled after the **TextIO** class from David Eck's Java programming textbook [7].

Operators. Bantam supports most Java operators including arithmetic operators (+, -, *, /, and %), relational operators (==, !=, <, <=, >, and >=), boolean operators (!, &&, ||), and assignment (=). Bantam also supports object construction, dynamic dispatch, object casting, and **instanceof**.

Memory management. Memory management in Bantam Java is similar to Java. Memory allocation occurs when objects are constructed (using **new**). Memory deallocation is handled implicitly, *i.e.*, a garbage collector is responsible for deallocating memory.

Excluded features. We left out many features that add little “bang for buck” to the compiler design such as modifiers, interfaces, and method overloading. We also left out other features whose inclusion would have made it difficult to incorporate in a one semester course. In particular, Bantam excludes the following Java features:

- Primitives besides **int** and **boolean**
- Arrays
- Static members
- Packages
- Modifiers such as **public**, **private**, *etc.*
- Nested classes
- Abstract classes
- Interfaces
- Final variables or methods
- Enumerations
- Generics
- Loops besides while loop
- Switch statements
- Logic operators (&, |, ^, <<, >>, >>>)
- Shortcut operations such as ++, *=, *etc.*
- Conditional expressions
- Hexadecimal/octal representation of **int** constants
- User-defined constructors
- Method overloading
- Extensive library for graphics, networking, *etc.*

A few of the features above would certainly add significant challenge to the project and might be doable in a one semester course. In future work, we will design an extended language for a more advanced compilers course that includes some of these features such as arrays, static types, and a floating point primitive.

5. BANTAM JAVA COMPILER

This section gives an overview of the Bantam Java compiler. The compiler is written in the Java programming language and will run on any x86/Linux machine (and perhaps other Unix-based machines). As shown in Figure 1, the compiler is split into four phases: lexical analysis, syntactic analysis, semantic analysis, and code generation. These phases are unimplemented; students must complete them in four course assignments.

Compiler phases. The lexer and parser are built using automatic lexer and parser generators (either JLex [5]/Java Cup [9] or JavaCC [12]). Students must implement the lexer and parser specification files. The semantic analysis and code generation assignments are somewhat more challenging. Students must build these almost from scratch given some auxiliary data structures (*e.g.*, symbol table). These two assignments will give students ample opportunity to make important design choices, such as how to build class hierarchy trees and class environments.

Abstract syntax tree. The compiler uses a single intermediate form: an abstract syntax tree (AST). Both the semantic analyzer and the code generator must traverse this AST in order to check the semantics of the program and generate code, respectively.

Auxiliary data structures. Several auxiliary data structures are provided for use in each of the compiler phases. These include (among others) a symbol table, an error handler, and AST nodes.

Packaging. For modularity purposes, the compiler relies heavily on Java packaging. Each compiler phase is placed in a separate package. (There are two packages for code generation, one for Mips and the other for x86.) In addition, a few other packages provide auxiliary support (*e.g.*, symbol table).

Visitor pattern. Again for modularity purposes, the source code also utilizes the visitor design pattern [8] for traversing a tree (*i.e.*, the AST) from an outside class. With the visitor pattern, AST traversals for checking the program semantics can reside within the semantic analysis package. Similarly, AST traversals for generating code can reside within the code generation package.

Runtime system. The Bantam compiler includes a runtime system, which handles memory allocation and deallocation (*i.e.*, the garbage collector) as well as all the methods



Figure 1: Bantam Java compiler phases. Note: AST is short for Abstract Syntax Tree.

of the built-in classes. The runtime system is provided to the student as it would take most students more than a single semester to implement on top of the rest of the compiler.

6. CONCLUSIONS

In this work, we present the Bantam Java language and compiler targeted specifically for use in an undergraduate compiler course. The Bantam Java language is a subset of the Java language, which allows students to focus exclusively on the compiler design and implementation rather than having to learn a new programming language. Students also can immediately apply what they learn to Java. Several aspects of the Bantam Java project are customizable (*e.g.*, target architecture, lexer and parser generator), allowing instructors to tailor the project to their own classroom needs. Finally, the Bantam Java project includes a free, comprehensive manual. Further information, including the lab manual, is available at <http://www.bantamjava.com>.

7. REFERENCES

- [1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley, 2nd edition, 2007.
- [2] A. Aiken. Cool: A portable project for teaching compiler construction. *SIGPLAN Notices*, 31(7):19–24, 1996.
- [3] A. Appel. *Modern Compiler Implementation in Java*. Cambridge, 1st edition, 1998.
- [4] A. Appel and J. Palsberg. *Modern Compiler Implementation in Java*. Cambridge, 2nd edition, 2002.
- [5] E. Berk. *JLex: A lexical analyzer generator for Java*, 1997. URL: <http://www.cs.princeton.edu/~appel/modern/java/JLex/current/manual.html>.
- [6] K. D. Cooper and L. Torczon. *Engineering a Compiler*. Morgan Kaufman, 2004.
- [7] D. Eck. *Introduction to Programming using Java - Part 1*. Creative Commons, 5th edition, 2006.
- [8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [9] S. Hudson. *Cup User's Manual*, 1999. URL: <http://www.cs.princeton.edu/~appel/modern/java/CUP/manual.html>.
- [10] J. Larus. SPIM S20: A MIPS R2000 simulator. Technical Report TR-966, Computer Sciences Department, University of Wisconsin-Madison, Sep. 1990.
- [11] T. W. Parsons. *An Introduction to Compiler Construction*. W H Freeman and Co, 1992.
- [12] Sun Microsystems, Inc. *JavaCC Documentation*. URL: <https://javacc.dev.java.net/doc/docindex.html>.
- [13] N. Wirth. *Compiler Construction*. Addison-Wesley, 1996.