



Bantam Java Compiler Project

Lab Manual

Version 1.1

Marc Corliss

Hobart and William Smith Colleges

corliss@hws.edu

<http://math.hws.edu/mcorliss>

E Christopher Lewis

VMWare

lewis@vmware.com

<http://www.eclewis.net/>

Draft Version

This is a draft version of the lab manual. We expect to complete the manual in summer 2008. If you encounter any problems or have suggestions for improvements, please email the first author at corliss@hws.edu.

Thank you for your patience,
Marc Corliss & E Christopher Lewis
9/6/2007

©2007, Marc L. Corliss and E Christopher Lewis

This manual is licensed under a Creative Commons Attribution-NonCommercial-NoDerivs 2.5 License (<http://creativecommons.org/licenses/by-nc-nd/2.5/>). This license allows you to duplicate and distribute this manual in unmodified form for non-commercial purposes. See the license for more details.

About the Authors

Marc Corliss is an Assistant Professor in the Mathematics and Computer Science Department at Hobart and William Smith Colleges. In 2006, Professor Corliss received his PhD from the University of Pennsylvania in computer science. His research interests are in system design, and in particular the design of compilers and processors. He is also interested in computer science education and building new tools for teaching computer systems courses.

E Christopher Lewis is a Staff Engineer in the Advanced Development Group at VMware, Inc. He is exploring new technologies in the implementation and application of virtual machines. Before joining VMware in 2007, Dr. Lewis was a Professor in the Department of Computer and Information Science at the University of Pennsylvania. He received his PhD in computer science from the University of Washington in 2001.

Acknowledgements

This work arose out of compiler classes taught at the University of Pennsylvania and Hobart and William Smith Colleges by both authors. First and foremost, the authors wish to thank the students and teaching assistants involved with those courses for their useful comments and feedback. The authors also thank two undergraduate students, Lori Pietraszek and Lex Kridler, both at Hobart and William Smith Colleges, for their contributions to this project. Lori designed and implemented an optimization component to this project. Lex built a sizeable set of library classes for Bantam Java. Although their work has not been added to the current infrastructure, it will be added in the next version (probably, summer 2008). Both students were supported by generous grants from the Provost's Office at Hobart and William Smith Colleges. The authors also wish to thank three institutions for their support: Hobart and William Smith Colleges, VMWare, and the University of Pennsylvania.

Contents

1	Introduction	1
1.1	Project Goals	3
1.2	Intended Use	4
1.3	Resources	5
1.4	Related Work	5
1.5	Future Work	6
1.6	Outline	7
2	Bantam Java Toolset	8
2.1	Overview and Installation Instructions	8
2.2	Using the Bantam Compiler	13
2.3	Last Words	16
3	Bantam Java Language	17
3.1	Overview	17
3.2	Inheritance	18
3.3	Class Definitions	20
3.4	Member Definitions	21
3.5	Statements	26
3.6	Expressions	31
3.7	Built-In Classes	41
3.8	Bantam vs. Java	45
3.9	Last Words	46
4	Bantam Java Compiler	47
4.1	Overview	47
4.2	Data Structures	48
4.3	Last Words	56
5	Bantam Java Runtime System	58
5.1	Runtime System Responsibilities	58

5.2	Encoding Objects and Primitives	59
5.3	Calling conventions	62
5.4	Last Words	64
6	Project 1: Building a Lexer	65
6.1	Overview	65
6.2	Files and Directories	65
6.3	Lexer Details	66
6.4	Testing the Lexer	67
6.5	Last Words	67
7	Project 2: Building a Parser	68
7.1	Overview	68
7.2	Files and Directories	69
7.3	Parser Details	69
7.4	Error Handling	71
7.5	Testing the Parser	71
7.6	Last Words	72
8	Project 3: Building a Semantic Analyzer	73
8.1	Overview	73
8.2	Files and Directories	74
8.3	Semantic Analyzer Details	74
8.4	Error Handling	77
8.5	Testing the Semantic Analyzer	77
8.6	Last Words	78
9	Project 4: Building a Code Generator	79
9.1	Overview	79
9.2	Files and Directories	79
9.3	Code Generator Details	80
9.4	Testing the Code Generator	82
9.5	Last Words	83

10 Closing Remarks	84
References	86

1 Introduction

This manual describes a new programming language called Bantam Java and a compiler infrastructure from which a student can build a Bantam Java compiler. The Bantam Java language and compiler are designed specifically for the classroom (*i.e.*, a compilers course). As the name suggests, Bantam Java contains a subset of the Java programming language, which is the language used in many computer science programs. As such, Bantam Java will seem familiar to most students. The compiler infrastructure, as well as the language, are designed so that an undergraduate student can complete the compiler in a one semester course.

Before describing the Bantam Java language and compiler, this manual motivates compiler construction for an undergraduate computer science student, which, unfortunately, some computer science programs have removed from their curriculum. There are many benefits in having an undergraduate student construct a compiler. First, constructing a compiler, the software responsible for translating high-level programs (*e.g.*, Java, C++ programs) into machine code, 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. Modern compilers are incredibly complex. 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 [11]. Implementing a compiler gives students experience applying all of these techniques. It also provides students with an opportunity to take part in designing a large software infrastructure.

A compiler is also 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 automata 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. In addition, a compilers course can help strengthen the understanding of those students who may have had some initial trouble with these often challenging concepts.

For all of these reasons, incorporating compiler construction into an undergraduate computer science curriculum is important. 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 are a number of programming languages and compilers [4, 16], which do not include some of the important features of modern 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, which has most of the important features of modern programming languages, but which can be designed and implemented in a one semester course.

This manual presents a new programming language (Bantam Java) and compiler infrastructure that is designed specifically for use in the classroom. It will run on any Linux/x86 machine. Bantam Java (or Bantam for short) is a small subset of the Java language, making it reasonable to implement a compiler for it in a one semester course. (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 (the document you are reading), which can be used in conjunction with any traditional compiler textbook [2, 17, 8, 14].

1.1 Project Goals

We had five goals in mind when designing the Bantam language and compiler.

Well-known language. Our first goal was to design a language that was a subset of Java, since Java is often used in introductory programming courses. We wanted to leverage student's existing intuitions developed in earlier programming courses. With a Java-based language, the instructor 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. Finally, the student spends less time learning the syntax and semantics of the language and more time focusing on the implementation of the language, and perhaps relearning some advanced language features, which they did not fully grasp the first time around.

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 would require 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 methods and variables, interfaces, abstract classes, nested classes, packages, exceptions, arrays, and most primitives (besides int and boolean). 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 comprehensive, free manual (the document you are reading) describing all aspects of the project. This manual will work well in tandem with any traditional compiler textbook [2, 17, 8, 14].

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 [7] and the Java Cup parser generator (an LALR parser generator) [12], or the JavaCC scanner and parser generator (an LL(k) parser generator) [6].

The infrastructure also supports two target machines: Mips and x86. The Mips target uses the SPIM emulator [13], which can be downloaded and installed on any x86 machine. The x86 target is a native target. It will work on any 32-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.

Summary. We made the Bantam language a subset of an existing, ubiquitous programming language, Java, so that students would already be familiar with it, understand the relevance of the projects, and be able to apply existing intuitions. Because the Bantam compiler had to be fairly simple, we could only choose a small subset of the Java features to include in Bantam. We chose to include those features that emphasize object-oriented programming. We use many tools from software engineering to make our infrastructure easy to understand and extend. We have written a comprehensive manual (this manual), which can be used in concert with a traditional compiler textbook [2, 17, 8, 14]. Finally, this project has some customizable components (lexer/parser generator, target machine) that allow the instructor to tailor the project to their own classroom needs.

1.2 Intended Use

This infrastructure and manual is intended for use as a set of course projects in a compilers course. It should be supplemented with a traditional compiler textbook [2, 17, 8, 14]. For example, this manual does not discuss the theory of lexing and parsing, nor does it discuss the compiler issues that arise when compiling languages besides Bantam. It also assumes that the student is following the design template laid out in the subsequent sections. It will offer limited help for those following a different design recipe.

In addition, this compiler project and this manual are intended for students who already know some Java. It does not teach programming or programming in object-oriented languages, although

it does discuss some of the concepts when describing the Bantam language. Students should be familiar with Java or at the very least familiar with a similar object-oriented language (*e.g.*, C++, C#). For those students who are less familiar with Java, they may need to consult a Java reference book [5], while reading this manual.

1.3 Resources

There are several resources for helping both students and instructors install and work with this toolset. They are all freely available online at <http://www.bantamjava.com>. These include:

- **Student manual.** A comprehensive student manual (this manual), which documents all aspects of the Bantam Java compiler project (a PDF document).
- **SIGCSE paper.** A condensed description of this work (a PDF document) published in SIGCSE 2008 [9]. The slides from the SIGCSE talk are also available (a PDF or PowerPoint document).
- **Bulletin board.** An online bulletin board for submitting questions, comments, and requests.
- **Compiler API.** The application programming interface (API) of the compiler generated from comments in the source code using the javadoc utility.

1.4 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 [4, 16], 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 by Alex Aikens [3] is one example of an object-oriented educational compiler project. This project uses a source language called COOL, a simple object-oriented language, which the authors are able to 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 significant 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 [17] is another educational, Java-based compiler project. The MiniJava project is integrated into a particular compiler textbook [17], 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.

Although Bantam Java is significantly different from Cool and MiniJava, and has the virtues discussed above, many ideas were borrowed from both projects. First, the context-free grammar for Bantam Java is specified in a similar syntax to the Cool context-free grammar [3]. The encoding of objects, object construction, and parts of the runtime system also work in a similar way as in Cool [3]. The Bantam Java compiler makes use of the visitor design pattern as in MiniJava [17].

1.5 Future Work

In future work, we will extend this project in the following ways:

- Extended language

We will add some extended language features such as arrays, static class members, and floating point primitives for use in more advanced compiler courses.

- Bantam Java library

We will build a small Bantam Java library, which will improve the utility of the language and make it more practical for writing large programs (*e.g.*, like what is described in the next bullet).

- Compiler written in Bantam Java

We will provide support for writing the compiler in Bantam Java, itself.

- Additional targets

We will add other targets such as the Java Virtual Machine.

- Additional course assignments

We will add some additional course assignments for use in a second semester compilers course or a graduate course. In particular, we will add assignments in code optimization and garbage collection.

1.6 Outline

The remainder of this manual is organized as follows. Section 2 describes the Bantam Java toolset and provides instructions for installing the toolset. Section 3 describes in detail the Bantam Java language. Section 4 describes the (incomplete) Bantam Java compiler, which students will complete in course projects. Section 5 describes the Bantam Java runtime system, which will be used by the generated code. Section 6, Section 7, Section 8, and Section 9 contain the project assignments and descriptions for building the lexer, parser, semantic analyzer, and code generator.

2 Bantam Java Toolset

This section gives an overview of the Bantam Java toolset as well as instructions for installing and running the compiler. Even if your toolset is already pre-installed you should still read (or at least skim) this section as it describes how the project infrastructure is layed out and how to run the Bantam compiler.

2.1 Overview and Installation Instructions

The Bantam toolset will compile and should run on any linux/x86 machine, although it is primarily tested on an Intel 32-bit x86 machine running Suse, Red Hat, and Ubuntu Linux. It also may run on other *nix platforms and other architectures. It is available as a compressed tarball (bantamjava.tar.gz) at the website: <http://www.bantamjava.com/>.

Once you have downloaded the tarball, move it to the appropriate location. To uncompress and untar the tarball, use the following:

```
bash$ tar xvfz bantamjava.tar.gz
```

This command will create a new directory, bantamjava (the *root directory*), in the current location. The bantamjava directory contains the following files and directories (note: directories end with '/', files do not):

```
README  Makefile  api/  bin/  lib/  man/  src/  tests/  tools/
```

Each of these files and directories are described below (although not in order).

README. The **README** file provides installation documentation as described in this chapter.

bin. The **bin/** directory contains executable files for running the compiler and other auxiliary tools. This directory is initially empty. Executable files are automatically added to **bin/** when the toolset is built.

lib. The **lib/** directory contains library files needed for building the compiler and also running auxiliary tools (*e.g.*, Spim). This directory is initially empty. Library files and directories are automatically added when the toolset is built.

man. The **man/** directory contains man pages for the Bantam compiler as well as some of the auxiliary tools (*e.g.*, Spim).

Makefile. The **Makefile** will compile and install all of the components of the toolset. There are

basically five parts of the toolset: the Javadoc web documentation describing the Java class files of the compiler (**api/**), the source files for the (incomplete) compiler (**src/**), the bantam test programs for debugging the compiler (**tests/**), and various tools used by the compiler (**tools/**). These components are described in more detail below.

To use the Makefile, you will need to edit one line of the Makefile. The definition of the variable **PROJPATH** (near the top of the Makefile) will probably be incorrect. Set this variable to the path of the installation directory. You may also want to edit a line just below the definition of **PROJPATH** that sets whether the compiler will be installed using the JLex and Java Cup lexer and parser generators or the JavaCC lexer and parser generator. The variable **BTMC** indicates which version will be installed. If it is set to “bantam” then JLex/Java Cup is used (the default), if it set to “bantam-jj” then JavaCC is used.

To make the entire toolset, type either “make” or “make all” in the root directory. You can also make an individual component as discussed below. To uninstall the toolset, type “make clean” in the root directory or in any of the subdirectories, which contain a **Makefile**.

api. The **api/** directory houses the Javadoc application programming interface documentation. After uncompressing and untarring the toolset, the directory will contain only a **Makefile**. The documentation is created from comments in the source code using the **javadoc** command provided with the Sun JDK [15]. It is placed in a directory **html/** within the **api/** directory. After generating the API (from the **Makefile**), it can be viewed online by copying this directory to a website. For example, if **/var/www/** is a website directory viewable at *http://www.foo.com/*, then the following commands would make the API viewable at *http://www.foo.com/bantam-api/*.

```
bash$ make api
bash$ cp -r api/html/ /var/www/bantam-api
```

When making changes to the compiler, Javadoc comments should be maintained so that any changes will be reflected in the API documentation. Sun Microsystems provides a free tutorial describing Javadoc commenting [15]. The API is also online at *http://www.bantamjava.com/api/*.

tools. The tools directory houses external software required by the Bantam Java toolset. These tools include JLex, a lexer generator [7]; Java Cup, a parser generator [12]; JavaCC, a lexer and parser generator [6]; and Spim, a Mips emulator [13]. These tools do not come with the toolset, but must be downloaded and moved to the **tools/** directory. Links to the respective websites are included on the *http://www.bantamjava.com*. These tools are built by typing “make tools” in the root directory or “make” in the **tools/** directory. This manual does not describe these tools (at

least the auxiliary tools) in much detail. For more information on these tools, check out their respective manuals [7, 12, 6, 13]. Feel free to use other installed versions of these tools. To use a different version of JLex or Java Cup, you will need to add symbolic links in `lib/` to the installation directories. For JavaCC, you will need to either add a symbolic link to the commands in `bin/` or change the `JAVACC` path variable in `src/bantam-jj/Makefile`. Spim is not a part of the build process so no changes are required.

Although the `tools/` doesn't initially contain the external tools, it does contain some code. First, it contains a directory `lib/` that holds two Jar files, which contain reference Bantam Java compilers: one that uses JLex/Java Cup and one that uses JavaCC. The `tools/lib/` directory also holds a runtime library file (containing assembly code) for both supported targets: Mips/Spim (`exceptions.s`) and x86/Linux (`runtime.s`). In addition to `lib/`, the `tools/` directory also contains a **Makefile** that will install each tool once they are downloaded, untarred and uncompressed, and moved to the `tools/` directory.

In order for the **Makefile** to work properly, you must use following names for each tool: `jlex`, `javacup`, `javacc`, and `spim`. Like the root directory **Makefile**, to use the **Makefile** in `tools/` you will need to edit one line of the **Makefile**. The definition of the variable, `PROJPATH`, on line 2 will probably be incorrect. Set this variable to the path of the installation directory. Once the tools have been downloaded and moved to the correct location, type 'make' to install them (Note: if compiled from the root directory the `PROJPATH` variable does not have to be changed. This change is only necessary when compiling from the sub-directory.)

Note: the tools do not have to necessary be installed within the `tools/` directory. They can be installed in other locations. However, a shell script or binary for running each tool must be installed in the `bin/` directory within the root directory. In addition, the `lib/` directory within the root directory must contain a link called `JLex` to the `JLex` directory within the installation of JLex. Also, the `lib/` directory must contain a link called `java_cup` to the `java_cup` directory within the installation Java Cup. Finally, the `lib/` some files within `tools/lib/` must be copied to `lib/`. These include the reference compilers (`bantamc-ref.jar`, which uses JLex/Java Cup and `bantamc-jj-ref.jar`, which uses JavaCC) and the runtime library assembly files for Mips/SPIM (`exceptions.s`) and x86/Linux (`runtime.s`).

src. The source files for the compiler are contained within `src/`. The Bantam compiler is written entirely in Java. Of course, the compiler is incomplete and will need to be completed by the student (although the code will compile). This directory contains two directories, `bantam/` and `bantam-jj/`.

The **bantam/** directory contains the source files that use the JLex lexer generator [7] and the Java Cup parser generator [12]. The **bantam-jj/** directory contains the source files that use the JavaCC lexer and parser generator [6]. The **bantam-jj/** directory uses some symbolic links to the bantam directory to prevent duplication of code. Students should implement the compiler within one of these directories (but probably not both) as specified by the course instructor.

Within both the **bantam/** and **bantam-jj/** directories are the following files and directories:

Makefile Main.java ast/ codegenmips/ codegenx86/ lexer/ parser/ semant/ util/ visitor/
Main.java contains the main class file for running the compiler. The lexer, parser, and semant directories contain Java packages for performing lexical, syntactic, and semantic analysis, respectively. The codegenmips and codegenx86 directories contain packages for performing code generation to the Mips and x86 (32-bit) architectures, respectively. Finally, ast, util, and visitor contain auxiliary class files required by the other packages. These packages are described in more detail in Section 4.

The **bantam/** and **bantam-jj/** directories share the ast, codegenmips, codegenx86, semant, util, and visitor packages. The actual files are contained within the bantam directory. The bantam-jj directory contains symbolic links to these directories.

The **Makefile** within **bantam/** or **bantam-jj/** will build the compiler. The **Makefile** in both **bantam/** and **bantam-jj/** depend on auxiliary tools (in the **tools/** directory within the root directory) and so the tools must be built before building the source code. Like the root directory **Makefile**, to use the **Makefile** in either **bantam/** or **bantam-jj/**, you will need to edit one line of the **Makefile**. The definition of the variable, **PROJPATH**, on line 2 will probably be incorrect. Set this variable to the path of the installation directory. To make the source in either the **bantam/** or **bantam-jj/**, type “make bantam” or “make bantam-jj” in the root directory. Alternatively, you can type “make” in either the **bantam/** or **bantam-jj/** directories. (Note: if compiled from the root directory the **PROJPATH** variable does not have to be changed. This change is only necessary when compiling from the sub-directory.)

As stated above, the Java code for the compiler is incomplete. In particular, the lexer, parser, semantic analyzer, and code generator contain only skeleton code. However, within each of these packages are working class files that will allow students to test their phases of the compiler in isolation. These files are backed up in files ending with “.ref” so if they are over-written or removed for some reason they can be recovered.

tests. The tests directory contains several Bantam programs that can be used to test the Bantam compiler. It must be built after building the compiler. Bantam programs end with the suffix “.btm”.

The tests directory contains several “.btm” files and a **Makefile**.

The source code for the compiler (in **src/**) must be built before building the test files. Once the source code is compiled and the program file **bin/bantamc** is built, then the test files can be compiled. To compile the Bantam programs, either type “make tests” in the root directory or “make” in the tests directory.

By default the compiler uses the Mips architecture with garbage collection disabled. To change either of these options, edit the **FLAGS** variable in the **Makefile**. Garbage collection is enabled by specifying the **-gc** flag. The x86 target is specified by using the **-t** flag. For example, the following compiles the Bantam programs for x86 with garbage collection enabled:

```
bash$ make FLAGS="-t x86 -gc"
```

Make produces assembly files for each Bantam program in the directory. The following will make only the assembly file **hello-world.s** from the Bantam program **HelloWorld.btm**.

```
bash$ make FLAGS="-t x86 -gc" hello-world.s
```

If the target is Mips (which is not the case above), then after building the assembly files we can run them using the Spim emulator [13] as follows:

```
bash$ ../bin/spim hello-world.s
```

If the target is instead x86, then we must first build a binary file from the assembly file and then run the binary file:

```
bash$ make hello-world
bash$ ./hello-world
```

This last step relies on having a working version of **gcc** [1] installed on your system and reachable from your **PATH** environment variable.

To make all x86 binaries, we can run the following command:

```
bash$ make FLAGS="-t x86" x86
```

Note: the command “make x86” (without setting **FLAGS**) will cause errors if the assembly files were generated for Mips.

The Bantam toolset includes a reference compiler for testing purposes. After implementing all of the compiler phases, the student’s compiler should work like the reference compiler. The reference compiler is built as part of the auxiliary tools (described above). After building the tools, the reference compiler is placed in **bin/bantamc-ref**. To run the reference compiler we could use the following:

```
bash$ ../bin/bantamc-ref HelloWorld.btm
```

We can also use the **Makefile** in **tests/** to compile the Bantam programs with the reference compiler. For example, the following will compile all of the Bantam programs with the reference compiler:

```
bash$ make BTMC=./bin/bantamc-ref
```

2.2 Using the Bantam Compiler

This subsection describes the command-line usage of the Bantam compiler. This material is also described in the Bantam compiler man page.

Although parts of the compiler are unimplemented (and will be implemented by the student), the main class of the compiler is implemented, which includes command-line, flag handling. So the unimplemented compiler will accept all options, although some may have no effect (*e.g.*, enabling debugging for the code generator).

The Bantam compiler accepts the following options:

```
bantamc [-o <output_file>] [-t <architecture>] [-gc] [-dl] [-dp] [-ds] [-dc]
        [-sl] [-sp] [-ss] <input_files>
```

Options. Each of the compiler options are enumerated and discussed below.

-o output_file

Specify the output file to use; the default is out.s

-t architecture

Specify the target architecture to generate code for; x86 and mips (must be all lowercase) are the currently supported target architectures. mips is the default.

-gc

Enable the garbage collector. The Bantam compiler uses a simple, mark-and-sweep garbage collector. By default the garbage collector is disabled.

-dl

Enable lexical analysis debugging. Note: students must add the debugging code that uses this flag. By default this flag is off.

-dp

Enable syntactic analysis debugging. Note: students must add the debugging code that uses

this flag (unless using the Java Cup implementation, in which case, debugging code is automatically inserted by the parser generator). By default this flag is off.

-ds

Enable semantic analysis debugging. Note: students must add the debugging code that uses this flag. By default this flag is off.

-dc

Enable code generation debugging. Note: students must add the debugging code that uses this flag. By default this flag is off.

-sl

Stop the compiler after lexing. If this flag is set, the compiler halts after performing lexical analysis. If lexical errors are discovered, these are printed to standard error, otherwise the scanned tokens are printed to standard output. By default this flag is off.

-sp

Stop the compiler after parsing. If this flag is set, the compiler halts after performing syntactic analysis. If lexical or syntactic errors are discovered these are printed to standard error, otherwise the parsed program is printed to standard output (as a Bantam source program). If -sl is specified, then this flag is ignored. By default this flag is off.

-ss

Stop the compiler after semantic analysis. If this flag is set, the compiler halts after performing semantic analysis. If errors are discovered these are printed to standard error, otherwise the annotated program (annotations indicate type information) is printed to standard output (as a Bantam source program with annotations in comments). If -sl or -sp is specified, then this flag is ignored. By default this flag is off.

Input files. The input files to the compiler must be Bantam files. These files should contain Bantam code (as discussed in Section 3) and end with the suffix “.btm”. If an input file is specified that does not end with “.btm” then the compiler will immediately print an error and exit. If the input files contain errors, then the compiler will not produce an output file, but instead print error messages and exit (unless the student has implemented a buggy compiler).

Output file. The output file is either a Mips or x86 assembly file depending on the `-t` flag (if `-t` is not specified then Mips is the target). Although not required, by convention this file should end with “.s”. By default, the compiler uses the file “out.s”.

Examples. Here are some example uses of the Bantam compiler. These examples assume we are in the `tests/` directory.

```
bash$ ../bin/bantamc -o foo.s Foo.btm
bash$ ../bin/spim foo.s
```

The first command above will compile the Bantam program `Foo.btm` and create a Mips assembly file (to be used with the Spim emulator [13]) called `foo.s`. The second command above runs the program using the Spim emulator for Mips.

```
bash$ ../bin/bantamc -t x86 -o foo.s Foo.btm
bash$ gcc -o foo foo.s
bash$ ./foo
```

The commands above compile and run `Foo.btm` on an x86, 32-bit machine. The first command compiles `Foo.btm` to `foo.s`, specifying the target as x86. The second command uses the `gcc` assembler [1] to translate the assembly file to an executable called `foo`. Finally, the third command runs the program. The x86 target uses the AT&T x86 assembly format, and must be run with an AT&T assembler, such as `gas`, the GNU assembler that is bundled with `gcc`. Currently, the x86 target can only be used on 32-bit x86 machines. In the future, we will add a target for 64-bit x86 machines.

```
bash$ ../bin/bantamc -gc -t x86 -o foo.s Foo.btm
bash$ gcc -o foo foo.s
bash$ ./foo
```

The commands above are identical to the previous set of commands (they compile and run `Foo.btm` on an x86 machine) except in this case the garbage collector is used.

```
bash$ ../bin/bantamc -sl Foo.btm
```

The command above performs only lexical analysis on `Foo.btm` and then prints out the scanned tokens to standard output, unless errors are discovered, in which case the errors are printed to standard error.

```
bash$ ../bin/bantamc -dp -sp Foo.btm
```

The command above performs lexical and syntactical analysis on `Foo.btm` and then prints out the parsed program to standard output, unless errors are discovered, in which case the errors are printed to standard error. In addition, debugging is enabled during parsing.

2.3 Last Words

You have now seen how to install and use the Bantam Java compiler. The upcoming sections will look in more depth at both the Bantam Java language and the Bantam Java compiler, starting first with the language.

3 Bantam Java Language

This section presents the Bantam Java programming language, the source language for the compiler. Bantam Java is a subset of the Java language. This section discusses those Java features that were included and excluded from the Bantam language. This section also describes in some detail the included features, although for a more detailed explanation one should consult a Java reference [5]. This section (and this manual) are intended for students who are already familiar with the Java programming language and its features.

3.1 Overview

Bantam, like Java, is an object-oriented programming language. As described in numerous textbooks [10, 5] (too many to cite all of them), an object is a self-sufficient entity that has some data and behavior associated with it, which is meant for modeling a real world entity such as a player, animal, room, *etc.*. To construct an object in Bantam or Java, one defines a class, which is essentially the blue print for all objects generated from that class. In other words, an object is an *instance* of a class. An object is effectively a variable and the class is the type of the object (although not always, as we discuss below), which defines both its data elements and its behavior.

A Bantam program consists of one or more files (with suffix “.btm”), each of which contains one or more class definitions. All code must reside within a class definition in Bantam. For this reason, classes and objects are critical and necessary components of any Bantam program.

Each class in Bantam, contains zero or more *members*. A member is either a *field* (part of the object’s data) or a *method* (part of the object’s behavior). Unlike in Java, all fields and methods must be instance variables and methods, that is to say they cannot be declared **static**. Each member is specific to a particular object; there are no class-wide members in Bantam.

When an object is created (the syntax and semantics of which are discussed later in the section) it is allocated its own copy of each field, and its field values may differ from the field values of other objects created from the same class. When an object’s method is called, it may use the object’s fields, therefore the outcome of a method depends largely on the owner object. Before an object is created, it is set to the special value **null**, which loosely speaking means the variable refers to an empty object.

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. Unlike in Java, the garbage collector is disabled by default and can be enabled using a compiler flag (see Section 2).

One of the classes defined in a Bantam program must be called **Main**. This class must contain a method called **main** (among possibly other members). When a Bantam program is started, a **Main** object is created and the **main** method is called. The program terminates when the **main** method finishes. The **main** method does all of the work of the program or calls methods that do the work on its behalf. Note that this starting point is different than in Java. In Java, which has static members, a static **main** method within a user-specified class is initially called.

In addition to classes, there are two primitive types in Bantam: **int** and **boolean**. An **int** holds a 32-bit signed integer. A **boolean** holds either **true** or **false**. Other primitive types from Java, such as **double** and **long** were not included in Bantam. These additional types add little intellectual challenge to the design and implementation of the compiler and for this reason were excluded from the language. As in Java, primitive types can be used in similar places as object types (*e.g.*, a field can have type **int**).

There are two types of comments in Bantam Java. The first is a single line comment, which begins with `//`. The text following `//` up to and including the end of the current line is ignored by the compiler (including `//`). The second type of comment is a multi-line comment that begins with `/*` and ends with `*/`. As the name suggests, these may span multiple lines. The compiler ignores all text in between `/*` and `*/` (including `/*` and `*/`). Multi-line comments may not be nested.

In the remainder of the section, we will look in more detail at defining classes and class members. But before doing that, we discuss a core feature of Bantam and most object-oriented languages, namely *inheritance*.

3.2 Inheritance

As in Java, Bantam allows classes to extend other classes. If class **A** extends class **B** then class **A** inherits all of class **B**'s members including all the members that **B** inherits from other classes. We say that **A** is a child of **B** and **B** is the parent of **A**. If a class does not explicitly extend another class then it automatically extends the built-in class **Object**, which is similar to the **Object** class from Java (albeit with less functionality). Bantam and Java support only single inheritance: a class inherits from exactly one other class (except the built-in **Object** class, which doesn't extend any

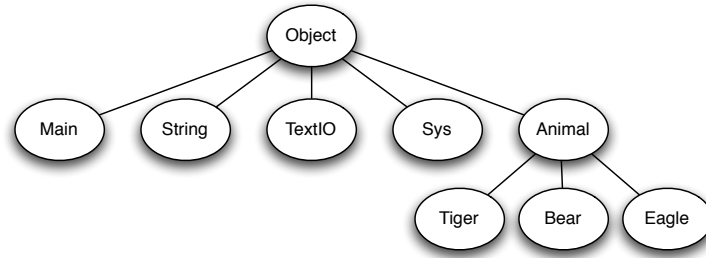


Figure 1: Example class hierarchy tree in Bantam.

other class).

To visualize the class dependencies, we can build a class hierarchy undirected graph where each node in the graph represents a class and edges in the graph exist between any two class nodes where one class extends the other. Because Bantam only supports single inheritance, the class hierarchy graph of a well-defined Bantam program is actually a tree with the class **Object** as the root.

Figure 1 shows an example class hierarchy tree in Bantam. Notice in the figure that the class **Tiger** is a descendant of the class **Object** (it extends **Animal**, which extends **Object**). We say that **Tiger** is a *subclass* of **Object** and that **Object** is a *superclass* of **Tiger**. More precisely, a class **A** is a subclass of a class **B** (or equivalently, **B** is a superclass of **A**) if we must pass through node **B** to get to the root of the class hierarchy tree from node **A**. Each class is defined to be a subclass and a superclass of itself. The class **Object** is a superclass of all classes and is only a subclass of itself.

When a class **A** is a subclass of a class **B**, we also say that **A** *conforms* to **B** since **A** must contain all of the functionality of **B**. Therefore, anywhere that we can use an object of type **B**, we can also use an object of type **A**. The opposite is not true. Because **A** may have more functionality than **B**, an object of type **B** cannot be used in place of an object of type **A**.

One implication of conformance is that the type of an object variable at compile time may differ from the type at runtime. For example, we can construct a new **Tiger** object and store it in a variable of type **Object** since **Tiger** conforms to **Object**. The compile time type of the variable, which is often called the *static type*, is **Object**, while the runtime type, which is often called the *dynamic type*, is **Tiger**. The dynamic type is often unknown until the program is run.

We will have more to say about inheritance and conformance in our discussion of class members (3.4) and casting (3.6).

3.3 Class Definitions

The format of a class definition is:

```
class <name> [ extends <parent> ] {  
    <members>  
}
```

where **<name>** is replaced with the name of the class, **<parent>** is replaced with the name of the parent class, and **<members>** is replaced with the class members (fields and/or methods). Note: the square brackets (‘[’ and ‘]’) are not part of the language, but are instead used to indicate that **extends <parent>** is optional. A class can explicitly extend another class by using **extends**. Alternatively, **extends** can be omitted, in which case the class automatically extends the built-in class **Object**.

Both **class** and **extends** are keywords in Bantam. Neither can be used as a name of a class, method, or variable. Bantam is case-sensitive, so **class** and **extends** are keywords, while **Class** and **EXTENDS** are not. The class name (**<name>**) and parent name (**<parent>**) are also case sensitive (in fact, all tokens in Bantam are case sensitive). Therefore, a class named **Animal** is not the same as a class named **aNimal**. The names may contain any sequence of upper or lowercase letters, digits, or the symbol ‘_’, although the name must start with a letter. By convention (taken from Java), class names generally start with an uppercase letter. If a class name contains multiple words combined into one name the start of each new word begins with an uppercase letter. For example, if we wanted a class name to include both “Bantam” and “Java” we would name the class **BantamJava**.

Unlike in Java, class definitions cannot use modifiers such as **public** and **package**. Since there are no packages in Bantam these modifiers are not necessary and using one is illegal. In addition, Bantam does not include abstract and static classes so the **abstract** and **static** modifiers are also illegal. Bantam also does not include interfaces so a class cannot use the keyword **implements** nor can an interface be defined. Finally, classes cannot be nested within other classes.

To define a (trivial) class **Animal**, we write the following:

```
class Animal {  
}
```

This class implicitly extends **Object**. It defines no data elements and no behavior. To do this,

we must define some class members.

3.4 Member Definitions

As stated above there are two kinds of class members: fields and methods. We look in turn at defining each, below.

Fields. The format for defining a field (*i.e.*, instance variable) is:

```
<type> <name> [ = <expression> ] ;
```

where **<type>** is replaced with the field's type, **<name>** is replaced with the field's name, and **<expression>** is replaced with the field's initialization expression. Note the '=' and initialization expression are optional. The **<type>** must be either a primitive type (*i.e.*, **int** or **boolean**) or a class name (*e.g.*, **Object**). The **<name>** is a non-empty sequence of upper and lowercase letters, digits, and the special symbol '_'. Each field defined within a particular class must have a distinct name.

By convention, field names generally start with a lowercase letter. Like class names, when multiple words are combined into one variable name, the start of each word (except the first word) begins with an uppercase letter. For example, if we wanted a field name to contain "Bantam" and "Java" we would call it **bantamJava**. The beginning lowercase letter indicates it's not a class name and the subsequent uppercase letters indicate the separation of words embedded within the name. If the field is meant to be constant (*i.e.*, never changed) then the convention is to use all uppercase and use '_' to indicate separation of words (*e.g.*, **BANTAM_JAVA**).

Below is another (slightly more interesting) definition of class **Animal**, which includes three field definitions:

```
class Animal {
    int numLegs = 4;
    boolean canFly;
    Animal mate;
}
```

The first field, **numLegs**, is a primitive variable with type **int**, which initially holds the value 4, although this can be changed later. The second field, **canFly**, is also a primitive variable, but with type **boolean**. It defines no initialization expression. When a field definition omits the

initialization expression, it is automatically assigned a value based on its type. Fields with type **int** are automatically assigned 0, fields with type **boolean** are automatically assigned **false**, and fields with object types are automatically assigned **null**. The third field is an object of type **Animal**. Its initial value is **null**, since it doesn't define an initialization expression. It has the same type as the class that is being defined, which is alright so long as we do not attempt to initially construct the object (object construction is discussed later in subsection 3.6). If we try to construct a new **Animal**, we will end up in an endless cycle of **Animal** object constructions. This problem only occurs for objects with types that are a subclass of the defined class.

We could have also written the following equivalent code:

```
class Animal {
    int numLegs = 4;
    boolean canFly = false;
    Animal mate = null;
}
```

It would not be equivalent to omit the assignment of **numLegs** to the value **4** since **numLegs** would have been automatically assigned **0**. The initialization expression can be arbitrarily complicated as discussed in 3.6. For example, we could have the following:

```
class Animal {
    int numLegs = 4;
    boolean canFly;
    Animal mate;
    int strength = (numLegs * 10) + 50;
}
```

The (contrived) initialization value for the fourth field, **strength**, is the initialization value of **numLegs** multiplied by 10, and added to 50 (*i.e.*, 90). Note that the initialization expression uses another field defined within the class. This is only legal if the referenced field is defined above the current field, since field initialization expressions are evaluated in the order in which they are defined in the class. Therefore the ordering of fields within the class matters. Also, the initialization expression is only evaluated once when the object is first created and used to assign a starting value to the field. If **numLegs** is changed after the object is created, the initialization expression for **strength** is not re-evaluated.

If a class extends another class, it can also reference any inherited fields:

```
class Tiger extends Animal {
    int tigerStrength = strength * 2;
}
```

Inherited fields are always evaluated before non-inherited fields, with the fields from the higher inherited classes (*i.e.*, closer to the root of the class hierarchy tree) evaluated earlier than fields from lower inherited classes (*i.e.*, farther from the root of the class hierarchy tree). When a **Tiger** object is created, the fields of **Object** are evaluated first, followed by the fields of **Animal**, and finally followed by the fields of **Tiger**.

Inherited fields in an extended class can be redefined. For example, we can do the following:

```
class Tiger extends Animal {
    int tigerStrength = strength * 2;
    Tiger mate;
}
```

The new field is treated as a separate instance variable, which essentially hides the inherited field within class **Tiger**. Notice that **mate** does not have to have the same type as the hidden inherited variable (or actually even a conforming type).

Every class implicitly has two special fields. The first variable, **this**, allows the programmer to refer to the current owner object. **this** can be used to pass the owner object to other methods. It can also be used to access members. The second, **super**, is similar to **this**, however, it only allows access to inherited members, not members defined within the class. In particular, **super** allows programmers to access hidden members. Neither **this** or **super** can be altered (it is illegal to assign to **this** or **super**), they are implicitly set when an object is created.

Unlike in Java, Bantam fields have no modifiers (*e.g.*, **public**, **static**). All fields are instance variables and are **protected**, and therefore field modifiers are not necessary (and are actually illegal). Fields can only be accessed from within the class or a subclass of the class. In Java, **protected** fields can be accessed from classes within the same package. Since Bantam does not include packages, **protected** fields cannot be directly accessed by any class that is not a subclass of the class where the field is defined. Accessing fields from other classes can only be done via methods (which are **public**). There is also no **final** modifier for indicating to the compiler that a field is constant. It is the responsibility of the programmer to guarantee that a constant field is never changed.

For reasons having to do with garbage collection, the maximum number of combined inherited and defined fields cannot exceed 1500. This restriction is not a part of the language (and could change in the future, although the maximum would only be made larger not smaller), but rather a part of the particular implementation of the language.

Methods. The format of a method definition is:

```
<type> <name> ( <parameters> ) {  
    <statements>  
    return [ <expression> ];  
}
```

where **<type>** is the return type of the method (*e.g.*, **int**), **<name>** is the name of the method, **<parameters>** is a list of comma-separated formal parameters to the method, and **<statements>** is a list of statements (possibly empty) terminated by semi-colons, which will be executed when the method is called. The last line in the method is always the return statement. This statement starts with the keyword **return** and can be followed by an optional expression. The return statement is terminated with a semi-colon. Like fields, each method defined within a single class must have a unique name (as discussed below, method overloading is not allowed). Method names follow the same conventions as field names. Below is an example of method **getStrength** in class **Animal**:

```
class Animal {  
    <field definitions (includes strength)>  
  
    int getStrength() {  
        return strength;  
    }  
}
```

The method **getStrength** takes no formal parameters and returns the value of the field **strength** (since the expression in the return statement is **strength**). This method is not contrived since fields are implicitly **protected** and can only be accessed from other classes via a **public** method. The method contains no statements beyond the final return statement. Because **strength** is an **int**, the return type of **getStrength** is **int**. Methods can also return nothing by declaring the return type as **void**. In this case, the return statement does not return an expression. Here is another method in **Animal**, which is **void**:

```

class Animal {
    <member definitions>

    void fight(int amount, boolean isWinner) {
        if (isWinner)
            strength = strength + amount * 5;
        else
            strength = 0;
        return;
    }
}

```

The method **fight** takes two parameters: **amount**, an **int** indicating the amount to fight, and **isWinner**, a boolean indicating whether the animal is the winner of the fight. The statement list contains an if statement, which will be discussed in Subsection 3.5, but for now it is enough to know that the statement increments the **strength** by 5 times **amount** if the animal is the winner, otherwise, it sets **strength** to 0. No value is returned (*i.e.*, no expression in the return statement) so the method must be declared **void**.

In Java, the **signature** of a method is the method name combined with the ordered list of parameter types. In the definitions above, the signature of **getStrength** is **getStrength()** and the signature of **fight** is **fight(int,boolean)**. Methods within the same class need only have a unique signature and not necessarily a unique name. This feature is called *method overloading*. In Bantam, method overloading is illegal and each method must have a unique name. However, fields and methods can share names. It is up to the compiler to determine, which is being referenced (the field or method) based on the context in which the name is used.

In the definitions of **getStrength** and **fight**, the type in the return statement matches the declared return type (**int** and **void**, respectively). However, that is only necessary when the declared return type is primitive or **void**. If the declared return type is instead an object type (*i.e.*, class name), then the type of the return expression only has to conform to the declared return type.

```

class Animal {
    <member definitions>

    Object getMate () {
        return mate;
    }
}

```

The variable **mate** was defined with type **Animal**, however, the method **getMate** is declared to

return type **Object**. This method definition is legal since **Animal** is a subclass of **Object** (although in this case we would probably want to specify **Animal** as the return type).

The **main** method in the **Main** class, which is the starting point of any Bantam program, must be declared **void** and define no parameters. Defining **main** in any other way is an error.

Unlike fields, there are no restrictions on the number of inherited and defined methods within a particular class. Like fields, methods do not have modifiers. All methods are **public**, meaning they can be called from within any class (given a dispatch object as discussed in 3.6). Like fields, methods can be redefined, however, a redefined method must have the same signature as the inherited method.

Let's now take a closer look at the statements that we can use within a method.

3.5 Statements

There are five types of statements in Bantam Java. We discuss each below.

Block statement. The first statement, a block statement, allows us to insert multiple statements where exactly one is needed. The format of a block statement is:

```
{ <statements> }
```

The block statement begins and ends with '{' and '}', respectively. Inside the braces are a list of statements. The utility of the block statement will become clear when we look at the other types of statements.

Declaration statement. A declaration statement declares a new variable and assigns it an initial value. This new variable is called a local variable since it may be accessed only within the method (there are other rules for accessing the variable, which are discussed below). The format of a declaration statement is:

```
<type> <name> = <expression> ;
```

where **<type>** and **<name>** are the type and name of the declared variable, respectively, and **<expression>** is the initialization expression of the variable. Declaration statements must be terminated with a semi-colon. The rules and conventions for declaration statements are similar to

fields except that the initialization expression is not optional. The initialization expression is not required in Java, however, Java requires that all local variables are assigned before they are used. Determining whether the variable is assigned before its first use requires some advanced compiler techniques, which we wanted to avoid in the Bantam compiler. We get around this by requiring all locally declared variables to be initially assigned.

Below is an example of a declaration statement:

```
class Main {
    void main () {
        int x = 3;
        ...
        return;
    }
}
```

In this case, we have declared a new variable **x**, which is local to the method **main**. The variable **x** cannot be used outside of the method **main**. In fact, it can only be used from the point it was defined until the end of the next enclosing block statement or the end of the method, whichever comes first. In this way, methods and block statements define new scoping levels. We can redeclare a variable so long as two variables with the same name are not declared in the same block statement or method. The redeclared variable hides the original variable until the end of the encompassing block statement or method. For example, in the code below the variable **x** is redeclared (note: the subscripts are not a part of the program, but instead, used to help the reader differentiate between the two variables):

```
class Main {
    void main () {
        int x1 = 3;
        int y = 0;
        int z = 0;
        {
            int x2 = 4;
            y = x2;
        }
        z = x1;
        return;
    }
}
```

The variable **x₁** is local to the method and the variable **x₂** is local to the inner block statement. The

assignment to **y** will use the second declaration of **x** (*i.e.*, **y** is set to 4), since the assignment resides after the second declaration and within the block statement. The assignment to **z** will use the first declaration of **x** (*i.e.*, **z** is set to 3), since the assignment resides after the first declaration (and the second declaration) and outside of the block statement where the second variable is declared. Declaration statements can also hide fields in addition to other local variables (although fields can still be accessed via **this**).

If statement. As in Java, an if statement allows the programmer to branch between statements based on some predicate. The format of an if statement is:

```
if ( <predicate> )
  <then statement>
[ else
  <else statement> ]
```

where **if** and **else** are keywords, the <predicate> is a boolean expression, the <then statement> is a statement that is evaluated if the predicate evaluates to true, and the <else statement> is a statement that is evaluated if the predicate evaluates to false. As in Java, the else statement is optional. Here is an example if statement:

```
class Main {
  void main () {
    if (animal.getStrength() > 100)
      animal.fight(100, true);
    else
      animal.fight(100, false);
    return;
  }
}
```

Only one statement can be used as the then or else statement. However, if we need to execute more than one statement in either of these (or both) we can use a block statement:

```

class Main {
    void main () {
        if (animal.getStrength() > 100) {
            animal.fight(100, true);
            strength = animal.getStrength();
        }
        return;
    }
}

```

If more than one predicate needs to be tested, if statements can be cascaded as they are in Java. For example:

```

class Main {
    void main () {
        if (animal.getStrength() > 100)
            animal.fight(100, true);
        else if (animal.getStrength() > 50)
            animal.fight(50, false);
        else if (animal.getStrength() > 10)
            animal.fight(10, false);
        else
            animal.fight(0, false);
        return;
    }
}

```

This code is actually made up of three if statements not one. The last two if statements are nested within the previous if statement's else clause.

Unlike Java, Bantam does not have switch statements, however, these can often be implemented using cascaded if statements like those in the code above.

While statement. A while statement can be used to repeat a statement some number of times. The format of a while statement is:

```

while ( <predicate> )
    <statement>

```

where <predicate> is a boolean expression, and <statement> is executed while the predicate evaluates to true. The while loop is evaluated by first evaluating the predicate, and if this is true then evaluating the statement, and repeating the process. The loop stops once the predicate evaluates to false.

Here is an example, while loop:

```
class Main {
    void main () {
        while (animal.getStrength() < 100)
            animal.fight(100, true);
        return;
    }
}
```

Like if statements, the statement within a while loop can be replaced with a block statement.

Bantam does not include any other types of loops such as for loops or do...while loops, however, these are easily converted into while loops. Below is a Java for loop and the corresponding Bantam while loop:

```
// Java for loop
for (int i = 0; i < 100; i = i + 1)
    animal.fight(10, true);
```

```
// corresponding loop in Bantam
int i = 0;
while (i < 100) {
    animal.fight(10, true);
    i = i + 1;
}
```

Expression statement. An expression statement is a statement made up of a single expression (which itself may contain subexpressions) and terminated by a semi-colon. Although we save the discussion of expressions until the next subsection, here is an example expression statement using a simple assignment expression (the variable on the left side of '=' is set to the computed value on the right side):

```
x = y + 2;
```

Not all expressions can be used within an expression statement. In fact, only two expressions are legal within an expression statement: assignments and method calls. The code below is not a legal expression statement:

```
foo() + 2;
```

Operator types	Operators
Arithmetic	+ - * / %
Relational	== != < > <= >=
Boolean	! &&
Assignment	=
Cast	(type)(expression)
Instanceof	instanceof
Dynamic dispatch	.
Object construction	new

Table 1: Bantam Java operators. The precedence and associativity is the same as in Java.

The addition of 2 to `foo()` is a useless operation in this case (since it isn't assigned to any variable), and for this reason, this is an illegal expression statement.

Let's now look at the last category of language constructs in Bantam: expressions.

3.6 Expressions

Bantam supports a variety of expressions, which are described below. Figure 1 lists the operators that are supported in Bantam. 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`. We discuss each of these in more depth below.

Every expression in Bantam computes a value with some data type. The type of the computed value is the type of the expression. Because expressions are used within field declarations, statements, or other expressions, the type of the expression is often important. For example, an expression that computes a value of type `int` cannot be used as the initialization expression for a `boolean` field. It is the responsibility of the compiler to determine the type of each expression and verify that it is used in the appropriate way. For each expression below, we will discuss both what the expression computes and the data type that it computes.

Assignment. An assignment expression has the form:

[<reference> .] <name> = <expression>

where <reference> is an optional reference object (followed by '.'), <name> is the name of

the target variable, and **<expression>** determines the value to assign to the variable. The target variable must refer to some variable defined in the same scope or an encompassing scope. The reference object can be used only in the assignment of a field. Because fields are **protected**, the reference object, if used, must be either **this** or **super**.

Here is an example assignment:

$$x = y + 2;$$

The expression on the right is evaluated first and then stored in the lefthand variable. The result of the entire expression is the result of the righthand expression.

If the type of the righthand expression or the lefthand variable is primitive then the types must match, otherwise the (object) type on the right must conform to the (object) type of the lefthand variable. The static type of the entire expression is the static type of the righthand expression.

Dynamic dispatch. A dynamic dispatch is simply a method call of an instance method. To call a method in Bantam, we need a **reference object** (of an appropriate type) to invoke the method on. The reason for the name *dynamic dispatch* is that in object-oriented languages a method call of an instance method is often thought of as passing a message to the object, hence the word “dispatch” (the “dynamic” part will be discussed shortly). Notice that because the called method can access fields, the outcome of the method depends on the reference object.

The format of a dynamic dispatch is:

$$[\text{<reference> . }] \text{<name> (<parameters>)}$$

where **<reference>** is an optional expression that corresponds to the reference object, **<name>** is the name of the method, and **<parameters>** are a list of comma-separated actual parameters, each of which is an expression. If the reference expression is not included, then **this** is used as the reference object.

In order for the dynamic dispatch to be legal, this method must be declared within the reference object’s static type class or within a superclass. The number of actual parameters and declared formal parameters must match and the type of each actual parameter expression must conform to the type of the declared formal parameter or match if either type is a primitive. Finally, the dynamic dispatch must be used according to its declared return type. For example, a **void** method

should not be used within a larger expression since it has no return value. A method that returns a **boolean** cannot be used in an arithmetic expression where an **int** is required.

The following code below shows how to dispatch on methods **fight** and **getStrength** within class **Animal**:

```
class Main {
    Animal animal = new Animal();

    void main() {
        animal.fight(20*5, true);
        int halfStrength = animal.getStrength() / 2;
        ...
        return;
    }
}
```

The dispatch on **fight** passes an **int (20*5)** as the first parameter and a **boolean (true)** as the second parameter, which match the declared formal parameters of **fight**. **fight** is a **void** method so it must appear on a line by itself and not within a larger expression. The method **getStrength** takes no parameters and returns an **int**. Therefore, we can use the result of **getStrength** in an expression that requires an **int**.

Because of inheritance, the exact method that is called on a dispatch is not always known at compile time, which is the reason for the word “dynamic” in dynamic dispatch. This property of object-oriented languages, *i.e.*, that one call site can call multiple methods during the course of a program, is known as *polymorphism*. For example, the class **Tiger** could define its own version of **fight**:

```
class Tiger {
    <member definitions>

    void fight(int amount, boolean isWinner) {
        if (isWinner)
            strength = strength + amount * 10;
        else
            strength = 0;
        return;
    }
}
```

If the method **fight** is called on an **Animal** object, which **fight** is called? It depends on the

runtime type of the reference object, which as discussed above may differ from the static type (*i.e.*, **Animal**). If at runtime, the object has type **Animal**, then the **fight** method within the **Animal** class is called. If, on the other hand, the object has dynamic type **Tiger**, then the **fight** method within the **Tiger** class is called. This example is illustrated in the **Main** class shown above. Although we know the original dynamic type of **animal** is **Animal**, the dynamic type of this field may change later, and in general, it is impossible for the compiler to determine the dynamic type of a variable. The compiler must generate code for calling the appropriate method at runtime based on the dynamic type of the reference object. A runtime error is generated if the reference object is **null**.

The reference expression can evaluate to any object type (so long as that class type includes the appropriate method) or can refer to the special variables **this** or **super**. Using **this** isn't necessary since excluding the reference object has the same effect. However, using **super** is often helpful. It allows the programmer to call hidden methods. For example, in the **Tiger** class, if we want to call the **fight** method from **Animal**, we can only do this using **super** as the reference object.

Object construction. We can use the **new** operator to create a new object from a class (called *object construction*). The format of a **new** operation is:

```
new <class name>()
```

where **new** is a keyword and **<class name>** is replaced with a class name. For example, if we wanted to create an object from a class **Tiger** and store it in a variable of type **Tiger** we could do the following:

```
Tiger t = new Tiger()
```

In Java, the programmer can define their own constructors that are called when the object is created. These allow the programmer to customize the object based on the particular context in which it is created. User-defined constructors are not allowed in Bantam. Object construction in Bantam creates a new object and initializes each field based on the initialization expression (or lack of). However, a Bantam programmer can write methods that simulate user-defined Java constructors. Of course, it is up to the programmer to make sure that the pseudo-constructor is

called after every object construction. For example, in the **Animal** class, we could define a pseudo-constructor to set the number of legs and the flag indicating whether the animal can fly.

```
class Animal {
    <member definitions>

    // simulates a constructor
    // must always be called immediately after construction
    void init(int l, boolean f) {
        numLegs = l;
        canFly = f;
        return;
    }
}

class Main {
    void main() {
        Animal animal = new Animal();
        animal.init(4, false);
        ...
    }
}
```

Casting. As we have already seen, we can store an object of one type into a variable whose type it is a super type. For example, we can store a **Tiger** object into a variable with type **Animal**. This action is called *upcasting*, since we are moving up the class hierarchy tree. Upcasting can be done either implicitly or explicitly. For example, the following two statements are equivalent:

```
Animal animal = new Tiger();
OR
Animal animal = (Animal)(new Tiger());
```

In the first assignment statement, we implicitly upcast a **Tiger** object into an **Animal** so that it can be stored in **animal**. In the second assignment statement, we do the same except we explicitly show the upcast. The format of an explicit cast is as follows:

```
( <class name> ) ( <expression> )
```

where **<class name>** is the name of some existing class and **<expression>** is an expression with an object type (*i.e.*, not a primitive). In Bantam, unlike in Java, the expression to be casted must be enclosed by parentheses (this simplifies parsing).

In the example cast above (*i.e.*, **(Animal)(new Tiger())**) only the static type of **animal** is **Animal**. The runtime type is still **Tiger**. The additional **Tiger** functionality (non-inherited members) is still maintained in the variable **animal**, however, it cannot be utilized. But it can be recovered by performing a *downcast*. For example, we could do the following:

```
Tiger tiger = (Tiger)(animal);
```

In this case, the cast must be explicit since it needs to be checked at runtime, *i.e.*, the compiler doesn't know statically whether **animal** is actually a **Tiger** or a subclass of **Tiger**. If the dynamic type of **animal** is not **Tiger** or a subclass of **Tiger** (*e.g.*, **Bear**) then an error occurs (and the program is halted).

Sometimes the compiler can statically catch an illegal cast (a cast that is neither an upcast or a downcast). In particular, any cast from one class to another class, which is neither a subclass or a superclass, is a compiler error. For example, the following is a compiler error:

```
Bear bear = (Bear)(new Tiger());
```

Upcasting and downcasting are common operations in Bantam (as well as in Java, prior to the addition of generics). Upcasting and downcasting make it possible to create collections of heterogeneous objects like the **Vector** class in Java. The collection is simply an object that stores each element as an **Object**, *i.e.*, objects are upcasted to **Object**. Because **Object** is a superclass of all classes, the collection can house any kind of object (but not a primitive). When an object is retrieved from the collection, it usually must be downcasted from **Object** to its original type. (Note: because the base Bantam language does not have arrays, collections in Bantam would need to rely on linked lists to store elements, which may not be all that efficient.)

Instanceof. The **instanceof** operation in Bantam allows the programmer to determine the runtime type of an object. Like casting, **instanceof** is often a useful operation when dealing with collections of heterogeneous objects. If we pull an object off of a collection, such as a **Vector**, we may not know its exact type, but rather that the object is one of several types. The format of **instanceof** is:

```
<expression> instanceof <class name>
```

where **<expression>** is an expression with an object type (*i.e.*, not a primitive) and **<class name>** is the name of some existing class.

The following code illustrates a common use of **instanceof**. The code retrieves a generic **Animal** object from a list (via the method **getNextAnimal**). The **instanceof** operator is used to determine the specific type of **Animal** (**Tiger**, **Bear**, or **Eagle**) so that a counter can be updated, which keeps track of the total number of each kind of **Animal**.

```
Animal animal = getNextAnimal();
if (animal instanceof Tiger)
    numTigers = numTigers + 1;
else if (animal instanceof Bear)
    numBear = numBear + 1;
else
    numEagles = numEagles + 1;
```

Arithmetic operators. Bantam supports five binary arithmetic operators (+, -, *, /, %) and one unary arithmetic operator (-). The format of the binary operators is:

<left expression> <operator> <right expression>

where **<left expression>** and **<right expression>** are both expressions with type **int** and **<operator>** is one of '+', '-', '*', '/', or '%'. The operators are similar to those in Java. **+** computes the sum of the left and right expressions, **-** computes the difference, ***** computes the product, **/** computes the integer division, and **%** computes the remainder when dividing the left expression by the right expression. The resulting type of these expressions is an **int**.

The format of the unary operator is:

- <expression>

where **<expression>** is an **int** expression. This operation computes the arithmetic negation of the expression. The resulting type is an **int**.

Here are some examples:

```

int x = 0;
int y = 1;
int z = 2;

x = y + z; // x is set to 3
x = y - z; // x is set to -1
x = y * z; // x is set to 2
x = y / z; // x is set to 0
x = y % z; // x is set to 1
x = - y; // x is set to -1

```

Boolean operators. Bantam supports two binary boolean operators (&& and | |) and one unary boolean operator (!). The format of the binary operators is:

```
<left expression> <operator> <right expression>
```

where <left expression> and <right expression> are both expressions with type **boolean** and <operator> is one of '&&' or '| |'. && computes the logical AND of the left and right expressions. If both the left and right expressions evaluate to **true**, then the result of && will also be **true**. Otherwise, the result is **false**. | | computes the logical OR of the left and right expressions. If both the left and right expressions evaluate to **false**, then the result of | | will also be **false**. Otherwise, the result is **true**. The resulting type of these expressions is a **boolean**.

The format of the unary operator is:

```
! <expression>
```

where <expression> is a **boolean** expression. ! computes the complement of the expression. If the expression evaluates to **false**, the result is **true**. If the expression evaluates to **true**, the result is **false**. The resulting type of these expressions is a **boolean**.

Here are some examples:

```

boolean b1 = false;
boolean b2 = false;
boolean b3 = true;

b1 = b2 && b3; // b1 is set to false
b1 = b2 | | b3; // b1 is set to true
b1 = !b2; // b1 is set to true

```

Relational operators. Bantam supports six binary relational operators (==, !=, <, <=, >, >=) for comparing two values. The format of the relational operators is:

<left expression> <operator> <right expression>

If <operator> is '==' or '!=' then the two expressions must have the same type if either is primitive or one type must conform to the other type if either is an object. If <operator> is '<', '<=', '>', or '>=' then the expressions must both have type **int**. The result of a relational operation is a **boolean** indicating whether the test succeeded.

The == operator tests if the left and right expressions are equivalent. Note: if the expressions are objects the operator tests if the left and right expression refer to the exact same object. If they refer to different objects then the test fails, even if the objects have the exact same field values. If the left and right expressions are equivalent then the result of == is **true**, otherwise it's **false**. The != operator works like == except that it tests if the left are right expressions are not equivalent. If the left and right expressions are equivalent then the result of != is **false**, otherwise it's **true**.

The < operator tests if the lefthand **int** is less than the righthand **int**, <= tests if the lefthand **int** is less than or equal to the righthand **int**, > tests if the lefthand **int** is greater than the righthand **int**, and >= tests if the lefthand **int** is less than or equal to than the righthand **int**. If the test succeeds then the result is **true**, otherwise the result is **false**.

Here are some examples:

```
boolean b = false;
int i1 = 0;
int i2 = 1;
int i3 = 0;
Object o1 = new Object();
Object o2 = new Object();
Object o3 = o1;

b = i1 == i2; // b is set to false
b = i1 != i2; // b is set to true
b = o1 == o2; // b is set to false
b = o1 == o3; // b is set to true
b = i1 < i2; // b is set to true
b = i1 >= i3; // b is set to true
```

Constants. There are three constant types in Bantam: **int**, **boolean**, and **String**. An **int** constant is

Operator	Associativity
()	non-associative
new	non-associative
.	left-associative
- (unary), !, cast	right-associative
*, /, %	left-associative
+, -	left-associative
<, >, <=, >=, instanceof	non-associative
==, !=	left-associative
&&	left-associative
	left-associative

Table 2: Operator precedence and associativity. Operators with higher precedence are listed higher in the table than those with lower precedence. Operators in the same row have the same precedence.

any sequence of numeric digits (*e.g.*, **0**, **124**). It must be between **0** and **2147483647** ($2^{31} - 1$) since it is stored as a 32-bit unsigned integer. Integer constants must be written in decimal, other forms such as hexadecimal are not supported. Leading 0's are allowed in an **int** constant.

A **boolean** constant is either **true** or **false**. These are keywords and may not be used as identifiers.

A **String** constant is a sequence of characters between double quotes (*e.g.*, "abc"). In Bantam, like Java, **String** constants are treated as objects of type **String** (which is discussed in 3.7). **String** constants may not span multiple lines. A backslash is interpreted as an escape character within a **String** constant, which allows the programmer to specify some special characters. These include `\n` (newline), `\t` (tab), `\"` (double quote), `\\` (backslash), and `\f` (form feed). Other special characters are illegal. In particular, a backslash within a **String** constant must be followed by 'n', 't', 'f', '\', or a double quote. For garbage collection reasons, the size of a **String** constant (and a **String** variable) is limited to 5000 characters.

Variables. A variable reference is also an expression. Of course, the variable must have been previously declared, otherwise, it's an error. If the variable is a field, then it can be preceded by **this** followed by `'.'`. If the variable is an inherited field (hidden or otherwise) then it can be preceded with **super** followed by `'.'`.

Parentheses. The final type of expression is an expression within parentheses: (`<expression>`). Parentheses are useful for grouping operations. Table 2 shows the order of operations and the

Class	Method signature	Method description
Object	Object clone()	copy an object
Sys	void exit(int status)	exit program with specified status
String	int length()	return string length
	boolean equals(String s)	test if strings are equivalent
	String substring(int beginIndex, int endIndex)	return substring between the indices
	String concat(String s)	return concatenated string
TextIO	void readfile(String filename)	set to read from specified file
	void writefile(String filename)	set to write to specified file
	void readStdin()	set to read from standard input
	void writeStdout()	set to write to standard output
	void writeStderr()	set to write to standard error
	String getString()	read next string
	int getInt()	read next int
	void putString(String s)	write specified string
	void putInt(int i)	writes specified int

Table 3: Bantam built-in classes. Note: **Sys**, **String**, and **TextIO** cannot be extended.

associativity for each operation discussed in this section. Parentheses should be used to override precedence and/or associativity. For example, we could do the following to force the addition to occur before multiplication:

```
int x = (2 + 3) * 5; // x is set to 25
```

3.7 Built-In Classes

There are four built-in classes in Bantam. These are listed in Table 3.

Object. The **Object** class is similar to the **Object** class in Java except with less functionality. All classes are a subclass of **Object**. **Object** contains one method called **clone**, which copies an object. If the cloned object contains fields (inherited or otherwise) then the values of these fields are copied to the new object. If these fields are themselves objects, however, the contents of those objects are not copied. In this case, the field in both the original and the copy would refer to the same object. To overcome this limitation, one can always redefine the **clone** method when defining a new class. The **Object** class is the only built-in class that can be extended, and in fact, all classes must either directly or indirectly extend **Object**.

Sys. The **Sys** class contains one method for exiting the program. It extends **Object**. The method **exit** takes one parameter, an **int** representing the exit status (0 is generally used for no error, non-zero values are used when an error has occurred). It replaces the static call to **System.exit** in Java. The **Sys** class cannot be extended.

String. The **String** class is a class for representing sequences of characters. It extends **Object** and cannot be extended. It is similar to the **String** class from Java, but with less functionality. For reasons having to do with garbage collection, the length of a **String** may not exceed 5000 characters.

String has four methods: **length**, **equals**, **substring**, and **concat**. The method **length** takes no parameters and returns the length (an **int**) of the **String**. For example, if the variable **s** is a **String** representing “abc”, then calling **s.length()** returns 3.

The method **equals** is a method for comparing two strings. It takes one parameter: a **String** to compare to the reference **String**. If the two strings represent the same sequence of characters then **equals** returns **true**. Otherwise, it returns **false**. Notice that this method behaves differently than **==**. The operator **==** can only determine if the two strings refer to identical objects. It cannot be used to determine if two strings represent the same sequence of characters. A runtime error occurs if the **String** parameter is **null**.

The method **substring** takes two parameters: a beginning index (an **int**) and an end index (an **int**). It returns the **String** that starts at the beginning index and ends at the end index. The character at the beginning index is included in the resulting **String**, but the character at the end index is not included in the **String**. Calling **s.substring(1,3)** returns a **String** representing “bc”. A runtime error occurs if the beginning index is less than 0, the end index is greater than the length of the **String**, or the beginning index is greater than the end index.

Finally, the method **concat** takes as parameter a **String** and returns a new string representing the concatenation of the reference **String** object with the parameter **String** object. For example, if we had a second **String s2** that represents “def”, then **s.concat(s2)** would return a new **String** representing “abcdef”. A runtime error occurs if the **String** parameter is **null** or if the resulting **String** has more than 5000 characters.

There is no method for copying strings, but because **String** extends **Object**, the **clone** method can be used to copy a **String**.

TextIO. The **TextIO** class provides methods for performing file input and output. It extends **Object**.

It is similar to the **TextIO** class defined by David Eck in his textbook [10], except the class does not use **static** members. Reading and writing are separately handled by this class and so it is possible with one **TextIO** object to read from one file, while writing to another. The **TextIO** class contains a method called **readFile** for setting the read file. This method takes as input a **String** representing the file name. It terminates the program if the file does not exist or the program does not have permission to read from the specified file. There is also a method called **writeFile** for setting the write file. This works similarly to **readFile** except the specified file need not exist. If the file does not exist, then **writeFile** will create the specified file. The program does, however, have to have permission to write to the specified file. For both **readFile** and **writeFile** the programmer can specify a path along with file name, *e.g.*, “./file.txt” or “/home/someuser/foo”. A runtime error occurs in either method if the input **String** is **null**.

TextIO also contains methods for reading from standard input (**readStdin**) and writing to either standard output (**writeStdin**) or standard error (**writeStderr**). These methods do not take any parameters. By default, **TextIO** reads from standard input and writes to standard output.

None of the **TextIO** methods described above actually read or write any data, instead they set either the read location or the write location. To read or write from some location, the programmer can use **getString**, **getInt**, **putString**, or **putInt**. The method **getString** reads the next line of text (minus the newline character) from the current read location and returns a **String** representing this text. The method **getInt** reads the next line of text (minus the newline character) from the current read location and interprets the text as an **int**. If an **int** was entered then **getInt** returns the entered value otherwise it returns 0. The method **putString** takes a **String** parameter and writes it to the current write location. A runtime error occurs in **putString** if the input **String** is **null**. The method **putInt** takes an **int** parameter and writes it to the current write location. Neither **putString** or **putInt** write a newline character to the output stream, however, this can be achieved by performing **putString(“\n”)** after writing the **String** or **int**.

Example. Below is a somewhat contrived example class that uses all of the builtin classes. There is one **Main** class with three methods: **main**, **getNextLine**, and **error**. The **main** method prompts the user via standard input for a positive integer **n** and then calls **getNextLine n** times to read **n** lines from the file “input.txt”. It stops early if the program reads the string “quit”. The method **getNextLine** reads the next line from “input.txt” and calls **error** if either there isn’t another line to read or if the length of the read string is less than 2. The method **error** prints an error message to

standard error and exits the program with status 1. After **main** gets the next line from **getNextLine** it concatenates it with the previous read output. At the end of the method, **main** prints the concatenated output to standard output and to the file “output.txt”.

```
class Main {
    TextIO io = new TextIO();
    String output = "";

    void error() {
        io.writeStderr();
        io.putString("Bad input; exiting\n");
        sys.exit(1);
        return;
    }

    String getNextLine() {
        String s = io.getString();
        if (s == null || s.length() < 2)
            error();
        return s.substring(1, s.length());
    }

    void main() {
        String s = "";
        io.readStdin();
        int n = io.getInt();
        if (n < 1)
            error();

        io.readFile("input.txt");
        int i = 0;
        while (i < n && !s.equals("quit")) {
            s = getNextLine();
            output = output.concat(s).concat("\n");
            i = i + 1;
        }

        io.writeStdout();
        io.putString(output);

        io.writeFile("output.txt");
        io.putString(output);

        return;
    }
}
```

3.8 Bantam vs. Java

Although we have described some of the differences between Bantam and Java throughout this Section, we lay out the main differences below. We also show how to convert a Bantam program to a Java program (note: converting a Java program to a Bantam program is a more difficult process, especially if the Java program uses a significant number of features that were excluded from Bantam).

Differences. Bantam excludes the following features (this list may not be complete):

- Primitives besides **int** and **boolean**
- Arrays
- Static members
- Packages
- Modifiers such as **public**, **private**, **protected**, and **package**
- 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
- Arbitrary return statement placement within a method
- Declaration statements without assignments
- Extensive library for graphics, networking, mathematical functions, *etc.*

Converting Bantam programs to Java programs. Because the Bantam language is a subset of the Java language, it is always straightforward to convert Bantam programs into Java programs. The programmer must perform the following steps:

- Add the **protected** keyword in front of all fields

- Add the **public** keyword in front of all methods
- Create classes TextIO and Sys in Java
- Rename the **main** method in the **Main** class to some unique name (*e.g.*, **oldMain**)
- Add the following static **main** method to the **Main** class

```
public static void main(String[] args) {  
    (new Main()).oldMain();  
    return;  
}
```

Otherwise, the program will not need to be changed.

3.9 Last Words

We have now looked closely at the Bantam Java language. As was discussed throughout this section, it is the responsibility of the Bantam Java compiler to enforce the lexical, syntactic, and semantic rules of the Bantam Java language layed out in this section. Therefore, this section will be an important resource when implementing the compiler. The next section gives an overview of the Bantam Java compiler, which must be completed by the student.



Figure 2: Bantam Java compiler phases.

4 Bantam Java Compiler

This section describes the Bantam Java compiler. This section first gives an overview of the compiler and then discusses the data structures that students will need to use in course projects.

4.1 Overview

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 2, 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 [7]/Java Cup [12] or JavaCC [6]). 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. Figure 4 lists the packages used in the Bantam Java compiler. Each compiler phase is placed in a separate package: `lexer`, `parser`, `semant`, `codegenmips`, and `codegenx86` (There are two packages for code generation, one for Mips and the other for x86.) In addition, three other packages provide auxiliary support (*e.g.*, symbol table): `ast`, `util`, and `visitor`. The `ast` package provides classes for representing

lexer	Performs lexical analysis
parser	Performs syntactic analysis
semant	Performs semantic analysis
codegenmips	Generates Mips code
codegenx86	Generates x86 code
util	Utility classes (symbol table, error handler, class tree node, location)
ast	AST node classes
visitor	Visitor classes (generic visitor, print visitor)

Table 4: Bantam compiler packages.

the nodes of the abstract syntax tree. The util package provides classes for error handling, for representing the class hierarchy tree, and for representing the symbol table. The visitor package (discussed below) provides classes for implementing the visitor design pattern [11].

Visitor pattern. Again for modularity purposes, the source code also utilizes the visitor design pattern [11] 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 built-in object methods. 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.

4.2 Data Structures

Below we describe the auxiliary data structures used by the compiler. This documentation is not complete, for example, we do not discuss every public method and field. This information is already available within the API documentation (see Section 2 for instructions on how to make the API documentation). Instead, we describe at a high-level how to use each data structure.

Error handler. The error handler data structure (**ErrorHandler**) allows for a unified way of handling and printing error messages from within any compiler phase. Each phase has an error handler object, which it uses to report errors. To report an error it must be registered with the error handler object by calling the **register** method. The **register** method does not immediately print the error, it is printed only after a call to the **checkErrors**. **checkErrors** prints out all errors (if there are any) and exits if there are any errors. Before printing the errors, **checkErrors** sorts the errors based on where

the errors were discovered, *i.e.*, the filename and line number. It first sorts the errors based on filename (using alphabetic order) and then sorts errors within the same file based on line number (using numeric order).

Class tree node. In order to compile a Bantam program, we must build a representation of the source program's class hierarchy tree (shown in Figure 1). The class tree node (**ClassTreeNode**) data structure represents each node (*i.e.*, a class) in the class hierarchy tree. Each class tree node contains a parent class tree node, which is **null** by default. It can and eventually should be set by calling **setParent** (only **Object** can have a **null** parent link). A class tree node also contains a list of child class tree nodes, which is empty by default. A child can be added by calling **addChild**. In general, only one of **setParent** and **addChild** needs to be called since both methods automatically set both the parent and child links. Each class tree node also has a variable and method symbol table associated with it, which can be retrieved by calling **getVarSymbolTable** and **getMethodSymbolTable**, respectively. These are the symbol tables for the particular class that the class tree node represents. Note: two symbol tables are needed since the variables and methods have separate name spaces. The class tree node also contains other meta information about the class, such as the class name and whether the class is extendable.

Location. The location data structure (**Location**) is a simple data structure for representing the location of a variable. It is used exclusively by the code generator (in concert with the symbol table) to record where a variable was allocated. The location data structure supports both variables allocated to memory and in a register. This information is needed during code generation. At a variable declaration, the variable name and location is added to the symbol table. At later references to the same variable, the location can be retrieved from the symbol table.

Symbol table. The Bantam compiler includes a symbol table (**SymbolTable**) for keeping track of all the identifiers and constants that appear within the source program. Unlike in commercial compilers, the symbol table in the Bantam compiler stores all symbols as strings. This simplifies the manipulation of symbols at the expense of a larger memory footprint.

The symbol table supports scoping, *i.e.*, nested name spaces. Given a symbol table object, the methods **enterScope** and **exitScope** will enter a new scope and exit the current scope, respectively. In Bantam, a subclass of another class has access to all of the inherited classes symbols. Rather than copying these symbols into the subclass, the symbol table has a **setParent** method for setting the parent symbol table. This method effectively nests the child class's scopes inside the parent

class's scopes.

The method **add** is used for adding a new name (a **String**) to the current scope. This method also takes a value (which can be any kind of object, *e.g.*, a **Location**), which can be retrieved later. For example, we might add the name "x" with value "int" (a **String**) to indicate that the variable "x" was declared with type "int". To lookup the value, we can use the method **lookup**, which takes as input a name. For example, we could call **lookup("x")**, which would return "int" (as an **Object**). If no entry is found with name "x", then **null** is returned. The method **lookup** looks for "x" in all scopes. To look in just the current scope we would use **peek**.

Abstract syntax tree. The Bantam compiler uses an abstract syntax tree [2, 17, 8, 14] (AST) as its intermediate representation. The AST is defined within the `ast` package. It has 51 different types of nodes, which are enumerated and discussed below.

ASTNode

An **ASTNode** is an abstract class that represents a generic node in the AST. It contains a line number indicating where the original source code (represented by the node) was scanned. All nodes in the AST extend **ASTNode**.

- **Program**

Each (correct) Bantam program contains one top-level **Program** node. A **Program** node extends **ASTNode** and contains a list of classes (**ClassList**).

- **ClassList**

A **ClassList** node extends **ListNode** and contains a list of **Class_** nodes.

- **Class_**

A **Class_** node represents a class from the Bantam source program. It extends **ASTNode** and contains a filename (**String**), a class name (**String**), a parent name (**String**), and a list of class members (**MemberList**).

- **Member**

A **Member** node is an abstract class that represents a class member in a Bantam source program. It is extended by the following:

- **Method**

A **Method** node represents a method from the Bantam source program. It extends **Mem-**

ber and contains a return type (**String**), a name (**String**), a list of parameters (**ExprList**), a list of statements (**StmtList**), and a return statement (**RetnStmt**).

- **Field**

A **Field** node represents a field from the Bantam source program. It extends **Member** and contains a declaration type (**String**), a name (**String**), and an (optional) initialization expression (**Expr**). For fields without initialization expressions, the initialization expression is **null**.

- **Stmt**

A **Stmt** node is an abstract class that represents a statement in a Bantam source program. It is extended by the following:

- **ExprStmt**

An **ExprStmt** represents an expression statement from the Bantam source program. It extends **Stmt** and contains an expression (**Expr**).

- **DeclStmt**

A **DeclStmt** represents a declaration statement from the Bantam source program. It extends **Stmt** and contains a declaration type (**String**), a name (**String**), and a (non-optional) initialization expression (**Expr**).

- **IfStmt**

An **IfStmt** represents an if statement from the Bantam source program. It extends **Stmt** and contains a predicate expression (**Expr**), a then statement (**Stmt**), and a else statement (**Stmt**).

- **WhileStmt**

A **WhileStmt** represents a while statement from the Bantam source program. It extends **Stmt** and contains a continuation expression (**Expr**) and a body statement (**Stmt**).

- **BlockStmt**

A **BlockStmt** represents a block statement from the Bantam source program. It contains a list of expressions (**ExprList**).

- **Expr**

An **Expr** node is an abstract class that represents an expression in a Bantam source program.

It is extended by the following:

- **AssignExpr**

An **AssignExpr** represents an assignment expression from the Bantam source program. It extends **Expr** and contains an optional reference object (**String**, which can be **null**), a lefthand variable name (**String**), and a righthand expression (**Expr**).

- **DispatchExpr**

A **DispatchExpr** represents a dynamic dispatch expression from the Bantam source program. It extends **Expr** and contains an optional reference object (**Expr**, which can be **null**), a method name (**String**), and a list of arguments (**ExprList**).

- **CastExpr**

A **CastExpr** represents an explicit cast expression from the Bantam source program. It extends **Expr** and contains a target object type (**String**), an expression to cast (**Expr**), and a flag (**boolean**) indicating whether it's an upcast or a downcast.

- **InstanceofExpr**

A **InstanceofExpr** represents an **instanceof** operation in a Bantam source program. It extends **Expr** and contains an expression to test (**Expr**) and an object type (**String**).

- **BinaryExpr**

A **BinaryExpr** is an abstract class that represents all binary expressions where the left and right operands are both expressions (*i.e.*, not casts or **instanceof**). It extends **Expr** and contains a lefthand expression (**Expr**) and a righthand expression (**Expr**). It is extended by the following:

- **BinaryArithExpr**

A **BinaryArithExpr** is an abstract class that represents binary arithmetic expressions. It extends **BinaryExpr**. It is subclassed by the following:

- **BinaryArithPlusExpr**

A **BinaryArithPlusExpr** represents an addition ('+') expression.

- **BinaryArithMinusExpr**

A **BinaryArithMinusExpr** represents a subtraction ('-') expression.

- **BinaryArithTimesExpr**

A **BinaryArithTimesExpr** represents a multiplication ('*') expression.

- **BinaryArithDivideExpr**
A **BinaryArithDivideExpr** represents a division (`'/'`) expression.
- **BinaryArithModulusExpr**
A **BinaryArithModulusExpr** represents a modulus (`'%'`) expression.
- **BinaryCompExpr**
A **BinaryCompExpr** is an abstract class that represents binary comparison expressions. It extends **BinaryExpr**. It is subclassed by the following:
 - **BinaryCompEqExpr**
A **BinaryCompEqExpr** represents an equivalence (`'=='`) expression.
 - **BinaryCompNeExpr**
A **BinaryCompNeExpr** represents a not equals (`'!='`) expression.
 - **BinaryCompLtExpr**
A **BinaryCompLtExpr** represents a less than (`'<'`) expression.
 - **BinaryCompLeqExpr**
A **BinaryCompLeqExpr** represents a less than or equal to (`'<='`) expression.
 - **BinaryCompGtExpr**
A **BinaryCompGtExpr** represents a greater than (`'>'`) expression.
 - **BinaryCompGeqExpr**
A **BinaryCompGeqExpr** represents a greater than or equal to (`'>='`) expression.
- **BinaryLogicExpr**
A **BinaryLogicExpr** is an abstract class that represents binary boolean logic expressions. It extends **BinaryExpr**. It is subclassed by the following:
 - **BinaryLogicAndExpr**
A **BinaryLogicAndExpr** represents a logical AND (`'&&'`) expression.
 - **BinaryLogicOrExpr**
A **BinaryLogicOrExpr** represents a logical OR (`'||'`) expression.
- **UnaryExpr**
A **UnaryExpr** is an abstract class that represents all unary expressions where the operand is an expression (*i.e.*, not **new**). It extends **Expr** and contains an expression operand (**Expr**). It is extended by the following:

- **UnaryNegExpr**

A **UnaryNegExpr** represents a unary integer negation ('-') expression.

- **UnaryNotExpr**

A **UnaryNotExpr** represents a unary boolean complement ('!') expression.

- **ConstExpr**

A **ConstExpr** is an abstract class that represents all constant expressions. It extends **Expr** and contains a constant value (**String**). It is extended by the following:

- **ConstIntExpr**

A **ConstIntExpr** represents an integer constant (*e.g.*, **12**) expression.

- **ConstBooleanExpr**

A **ConstBooleanExpr** represent a boolean constant (*e.g.*, **true**, **false**) expression.

- **ConstStringExpr**

A **ConstStringExpr** represent a string constant (*e.g.*, "**abc**") expression.

- **VarExpr**

A **VarExpr** represents a variable reference expression. It extends **Expr** and contains an optional reference object name (**String**, which might be **null**) and a variable name (**String**).

- **ListNode**

A **ListNode** is an abstract class that represents a generic list of nodes in the AST. All lists of nodes must extend **ListNode**. These include the following:

- **MemberList**

A **MemberList** node extends **ListNode** and contains a list of **Member** nodes.

- **StmtList**

A **StmtList** node extends **ListNode** and contains a list of **Stmt** nodes.

- **ExprList**

An **ExprList** node extends **ListNode** and contains a list of **Expr** nodes.

For more information on the API of each different class of AST node, view the API documentation at <http://www.bantamjava.com/api/>.

Visitor pattern. The last data structure is the Visitor design pattern [11]. The Visitor pattern allows us to traverse a tree (such as an AST) from a separate class. Before describing how it works, let's look at why we need it.

In order to implement the various phases of the compiler (in the upcoming sections), it will be necessary to traverse the AST several times. For example, to print the AST (if the compiler option “-sp” or “-ss” is set), we must walk over each AST node and print out a representation of it. In Java, there are essentially two ways we can do this traversal. We will first discuss a naïve approach and then discuss a better alternative. One approach is to add a method to each AST node that when called will print that particular node. The problem with this approach is that we have to add a method to every class in the ast package for printing that particular node. If printing was the only reason for traversing the AST then this solution might be acceptable. However, we will also need to traverse the AST to type check the program and to generate assembly code (and these might be further broken down into multiple traversals). Therefore, we would have to add a lot of code to each of the classes in the ast package (all 51 of them!). Furthermore, in this approach, we are mixing functionality into the same classes. The AST classes should only contain code for representing a node in the abstract syntax tree. It should not contain code for printing the AST, or performing semantic analysis, or performing code generation. Imagine that we used this approach, and we wanted to make a small change to the semantic analyzer. This change could require making modifications to each class in the ast package!

A better alternative is to put the traversal code in a separate class. If we are traversing the AST in order to type check it, then that class would be a part of the semant package. But care needs to be taken in designing this traversal code. For example, many AST nodes contain sub-nodes whose exact dynamic type is unknown. An if statement has a predicate sub-node, which has static type **Expr**. This sub-node could actually be an **instanceof** expression, a **==** expression, or any other type of expression. We could potentially have some large nested case statements that perform the appropriate actions based on the dynamic type of each sub-node. This would quickly become far too error-prone and nearly impossible to maintain.

Instead, we can use the Visitor pattern, which works as follows. We first build an abstract class called **Visitor**. Every class that needs to traverse the AST extends **Visitor**. The **Visitor** class defines a **visit** method for each type of AST node. For example, here is the **visit** method for **BinaryArithPlusExpr**:

```

public Object visit(BinaryArithPlusExpr node) {
    node.getLeftExpr().accept(this);
    node.getRightExpr().accept(this);
    return null;
}

```

This method gets the left and right expressions and calls an **accept** method on each. This **accept** method must be added to each class in the ast package, however, it only has to be done once, rather than each time we want to do a different traversal. The **accept** method works as follows:

```

public Object accept(Visitor v) {
    return v.visit(this);
}

```

By calling **visit** and then **accept**, we can vector back into the appropriate **visit** method for the sub-node. For example, if the left-hand expression of the **BinaryArithPlusExpr** were a **ConstIntExpr**, then after calling **visit** on the left-hand expression, and then **accept** in **ConstIntExpr**, we would end up in the method **visit(ConstIntExpr node)**.

Essentially, we are simulating multiple dispatch with single dispatch. We need to call a method based on both the runtime type of a node in the AST as well as a particular type of visitor. Since Java only supports single dispatch, we can achieve this by first dispatching into the particular AST node, and then dispatching back into the visitor object.

To build a new visitor class (for example, one that performs type checking), you will need to extend **Visitor**. You can then override each **visit** method with a method that traverses the AST and performs the additional functionality (*e.g.*, type checking). As stated earlier, you will need to do this when type checking the program during semantic analysis and when generating code during code generation. In fact, each phase, *i.e.*, semantic analysis and code generation, may require multiple visitors. To view an example visitor, look at the **PrintVisitor** class in the visitor package, which traverses the AST and prints each node. Your visitors will look structurally similar to **PrintVisitor**.

4.3 Last Words

This section has given an overview of the Bantam Java compiler, which translates Bantam Java source code into assembly code. The student is responsible for implementing a working compiler,

which must follow the language specifications layed out in the previous section and must use the data structures and code structure layed out in this section. In addition to these requirements, the assembly code generated by the student's compiler must interoperate with the runtime library code for handling memory allocation and deallocation, and built-in class methods, among other things. This is the topic of the next section.

5 Bantam Java Runtime System

This section describes the Bantam Java runtime system. The runtime system is provided with the Bantam Java compiler in the `lib/` directory within the main Bantam Java installation directory. If you are generating code for Mips/Spim then the runtime system is contained within the file `exceptions.s`. If you are generating code for x86/Linux then the runtime system is contained within the file `runtime.s`. The runtime system contains assembly code for performing various tasks such as garbage collection and error handling (among others). In order to use the runtime system (*i.e.*, if you do not want to have to write your own runtime system), your compiler must follow a specific format for encoding objects and primitives in memory. In addition, your code generator must use the same calling conventions.

Below we discuss each of the responsibilities of the runtime system, the format for encoding of objects and primitives, and the calling conventions.

5.1 Runtime System Responsibilities

The runtime system contains assembly code for initiating the program, performing memory management, executing the built-in methods (*e.g.*, **Object.clone**, **String.concat**), and for error handling.

Program initiation. The compiled program begins executing in an initial subroutine within the runtime system (`_start` for Mips, `main` for x86). This subroutine does some initialization of runtime system data structures, and then calls the `main()` method within the `Main` class (which is assembly code compiled by your compiler, *i.e.*, it's not part of the runtime system).

Memory management. The runtime system also contains several subroutines for allocating and deallocating memory. Deallocation is done via a simple, mark-and-sweep garbage collector (unless garbage collection is disabled). None of the methods have to be called by the compiled code, they are called automatically every time an object is created (below, we discuss how to create objects).

Built-in methods. The runtime system also contains the assembly code for every method of a built-in class (methods within **Object**, **Sys**, **String**, and **TextIO**). Your compiler should not generate code for these methods.

Error subroutines. Finally, the runtime system contains subroutines for handling errors. Most error handling subroutines are called automatically by code within the runtime system. There

<code>._null_pointer_error</code>	Null pointer referenced	No additional arguments
<code>._class_cast_error</code>	Class cast error	Object ID passed in t0 (Mips) or ebx (x86), target ID passed in t1 (Mips) or ecx (x86)
<code>._divide_zero_error</code>	Divide (or mod) by zero	No additional arguments

Table 5: Bantam Java runtime system error subroutines.

are three that must be called by your compiled code. These are shown in Figure 5. All three of these subroutines take as input the current line number and a pointer to a string representing the filename. For a Mips target, the line number is passed via the \$a1 register and the filename pointer is passed via the \$a2 register. For the x86 target, the line number is passed by storing it to the memory address, `._current_line_number` and the filename pointer is passed by storing it to the memory address, `._current_filename_ptr`. Both `._current_line_number` and `._current_filename_ptr` are defined within the runtime system. The line number and filename must also be set when dispatching to a method or when constructing an object, since either of these operations could result in a call to an error subroutine. In addition, the error subroutine `._class_cast_error` takes two additional arguments: the numeric identifier for the object in the cast expression and the numeric identifier for the target class in the cast expression. For Mips, the object identifier is passed via register \$t0 and the target class identifier is passed via register \$t1. For x86, the object identifier is passed via register ebx and the target class identifier is passed via register ecx.

5.2 Encoding Objects and Primitives

To use the runtime system, objects and primitives must be encoded using the format discussed below.

Primitives. There are two types of primitives in Bantam Java: ints and booleans. Nothing special needs to be done for ints. To generate a word (in assembly) with the value -10, the compiler would generate the following directive:

```
.word -10
```

Assembly language does not have support for booleans, so true and false must be encoded using numeric values. False is encoded using 0 and true is encoded using a non-zero value. Probably the best non-zero value to use for true is -1 (we leave it to the reader to figure out why).

Objects. Figure 3 shows the encoding of an object. An object is encoded as a table with the following entries: (1) a numeric identifier of the class that the object was constructed from (each

Numeric identifier
Size in bytes
Dispatch table pointer
Value of field 1
Value of field 2
...
Value of field n

Figure 3: Bantam Java encoding of an object with n fields.

class's identifier is unique), (2) the size of the table in bytes, and (3) a pointer to the dispatch table (which are discussed below). The table also contains an entry per field defined (inherited or otherwise). If there are n fields then there are n words that follow the dispatch table pointer, each of which contains the value for the i th field. If a class has no fields then the dispatch table pointer is the last word in the object. In this case, the size of the object is 12 bytes. In general, for an object with n fields (inherited or otherwise), the size of the object is $12 + n * 4$.

Strings. Strings have a slightly different format than other objects. This is because unlike other objects, the size of string objects can vary. The size of the object depends on the length of the sequence of characters. Like other objects, a string object contains an identifier, size, and dispatch table pointer. The fourth entry is the length of the string (in number of characters). Following that is a byte holding each character encoded in ASCII, followed by a null terminating byte (*i.e.*, 0), and possibly followed by other null bytes so that the entire object is word-aligned. The assembly code in Mips for encoding the string "abcd" would look as follows:

```
.word 2           # String identifier
.word 24         # size of object in bytes
.word String_dispatch_table # pointer to String's dispatch table
.word 4         # length of the string
.byte 97        # ASCII encoding for 'a'
.byte 98        # ASCII encoding for 'b'
.byte 99        # ASCII encoding for 'c'
.byte 100       # ASCII encoding for 'd'
.byte 0         # null terminating byte
.byte 0         # alignment byte
.byte 0         # alignment byte
.byte 0         # alignment byte
```

Note: if the ISA supports the directives ".asciiz" and ".align" (both Mips and x86 do), then these can be used to simplify the code as follows:

```
.word 2           # String identifier
```

```

.word 24                # size of object in bytes
.word String_dispatch_table # pointer to String's dispatch table
.word 4                 # length of the string
.asciiz "abcd"         # ASCII representation of "abcd", followed by null byte
.align 2               # add appropriate number of 0 bytes to word align

```

Class name table. Your compiler will also need to generate a table that the runtime system uses for error reporting. Some error subroutines in the runtime system need to print class names given an object (*e.g.*, on a class cast error). To find the name, it uses a special table called **class_name_table**, which has an entry for each class (built-in or user-defined). The table is indexed by the class identifier. Each entry contains a pointer to a string representing the name of the class. Your compiler will have to generate the **class_name_table** as well as the strings for each class name (including strings for built-in classes).

Object references and null. Objects are referred to using their memory address. An object reference (*e.g.*, a variable) is simply a pointer in memory (a word) that contains the memory address of the object that it refers to. If an object reference is null, then the pointer contains the value 0. Therefore, testing for null is achieved by comparing the reference to 0.

Dispatch tables. For each class, your compiler must generate a dispatch table (including for built-in classes). The dispatch table holds the addresses of every method that is visible from within that particular class. These must be named using the following format:

```
<class name>_dispatch_table
```

Each word in the dispatch table contains the memory address of the corresponding method. There is an entry for each method that is visible by an object generated from that class (inherited or otherwise). Order within the dispatch table is important. Methods from classes higher in the class hierarchy tree (*i.e.*, closer to **Object**) should appear earlier in the dispatch table. The order of methods defined within a particular class should be fixed for each dispatch table. In the absence of redefined methods, the order is the same as the order in which the method was defined within the class. If an extended class redefines a method, then the address of the redefined method is placed in the entry that originally contained the inherited method. A new entry is not created. This last requirement is important for handling inheritance and object casting. If an object is casted to a super type, and a method is called that was overridden in the extended class, then this encoding ensures that the overridden method is the method that is called.

Object templates. Objects are not dynamically constructed from scratch. Instead, they are con-

structed from object templates. For each class, your compiler will generate an object template (including for built-in classes). These must be named using the following format:

`<class name>_template`

For example, your compiler would generate a template for the class **Object** called **Object.template**. When a new object is requested (using **new**), the template for the corresponding class is copied using the built-in subroutine `Object.clone` (provided in the runtime library). The templates contain default values for each field in the class (0 for ints, false for booleans, null otherwise).

Object initialization. Your compiler will also need to generate subroutines for initializing an object based on the initialization expressions of each field. These must be named using the following format:

`<class name>_init`

These will execute each initialization expression in the order that they appear within the class, and assign the result to the corresponding field. During object construction, this subroutine should always be called after the object is cloned (although it should not be called if clone was called directly by the Bantam Java program).

Example. Figure 4(a) shows part of a contrived program. Figure 4(b) shows graphically three of the object templates and three of the dispatch tables that are generated by the compiler (note: other templates and dispatch tables would also be generated). Note that the method **method2** in class **Foo** is overridden in class **Goo**. It appears in the same place in the dispatch table in the **Goo** dispatch table as it does in the **Foo** dispatch table, however, in the **Goo** dispatch table it points to the method in **Goo** rather than in **Foo**. Note also that the field **field1** is redefined in class **Goo**. With fields, the redefined field does not replace the existing field. Instead a new entry is made for the redefined field. When executing code in class **Goo**, the second entry is used when **field1** is referenced. Alternatively, when executing code in class **Foo**, the first entry is used when **field1** is referenced.

5.3 Calling conventions

The Bantam Java runtime system uses precise calling conventions that must be followed in order for the compiled program to work correctly. These conventions are discussed below. Note that these conventions only apply to Bantam Java methods. The calling conventions for error subrou-

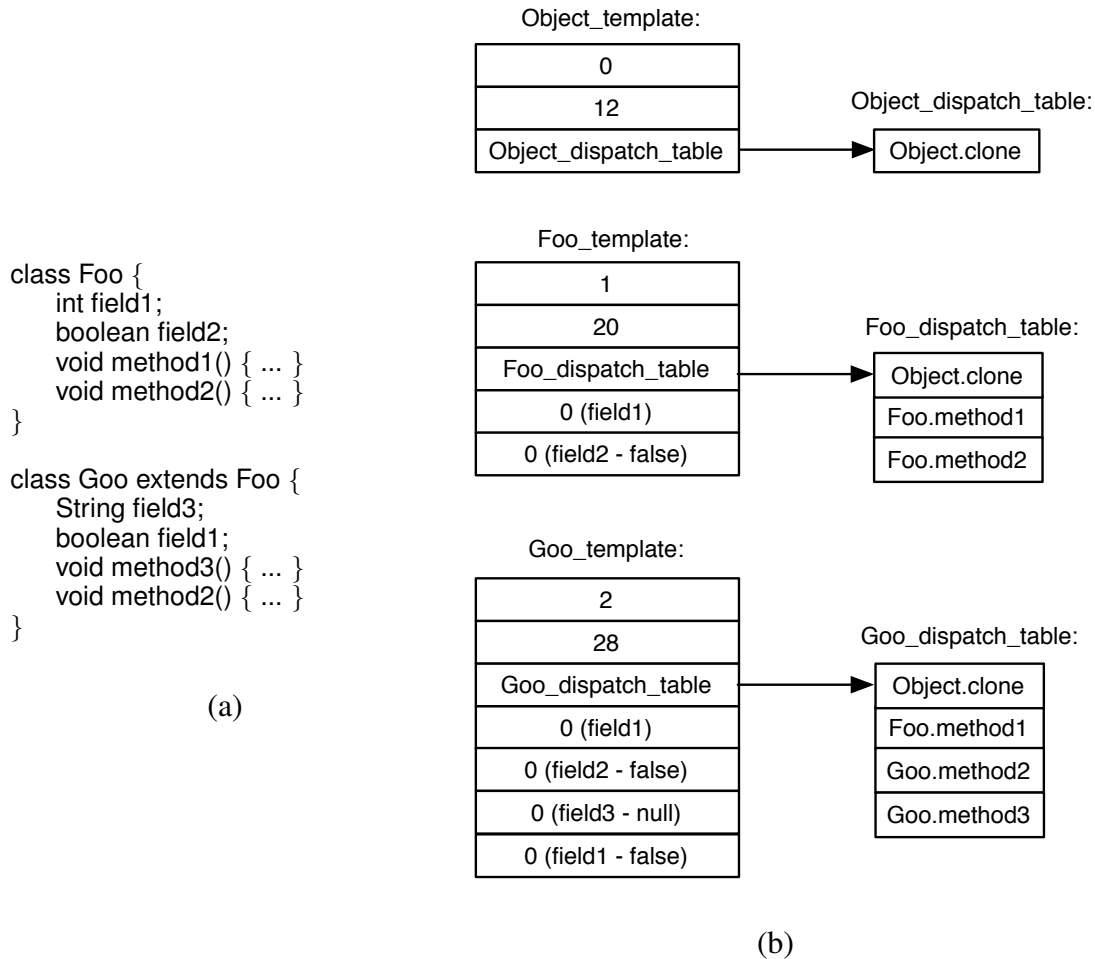


Figure 4: Some example templates and dispatch tables. (a) shows some Bantam Java code (which is incomplete) and (b) shows the templates and dispatch tables for three of the classes: Object, Foo, and Goo.

tines are discussed above in Section 5.1.

Passing the reference object to the callee. Because Bantam Java does not support 'static' all methods calls are dispatches using a reference object. The reference object is an implicit argument and must be passed to the called method. As in Java, Bantam Java supports only call by value. For objects, it is the reference to the object that is copied not the object itself. In Mips this address is passed in the register \$a0 and in x86 this address is passed in the accumulator register eax.

Passing parameters to the callee. All other arguments are passed via the stack. Starting from the leftmost argument and proceeding to the rightmost argument, each actual parameter is evaluated

and the result is pushed onto the stack. To push it onto the stack, the value is stored to the address held in the stack pointer and the stack pointer is decremented by one word (x86 actually has an instruction for performing a push).

Passing the return value to the caller. The return value (if there is one) is passed via a register. In Mips it is passed via register \$v0 and in x86 it passed via the accumulator register eax.

5.4 Last Words

Over the past three sections, we looked at the Bantam Java language, compiler, and runtime system. You are now ready to write your very own compiler. In the next four sections, you will build the four components of the Bantam Java compiler: a lexer (Section 6), parser (Section 7), semantic analyzer (Section 8), and code generator (Section 9), respectively.



Figure 5: Bantam Java compiler phases. The lexer phase is shaded.

6 Project 1: Building a Lexer

In the first project, you will build a lexer using an automatic scanner generator. You will need to complete the code within the lexer package of your compiler source code. Make sure that you have read Section 3 on the Bantam Java programming language and Section 4 on the Bantam Java compiler before starting this project.

6.1 Overview

The lexer is the phase of the compiler that builds tokens from a stream of characters. Figure 5 shows the lexer (shaded) along with the other compiler phases. The lexer matches on all keywords, identifiers, constants, and special symbols (*e.g.*, ‘;’) in the program and builds a token for each match that is passed on to the parser for syntactic analysis. For example, if the lexer scans the text “99 “ (without the quotes), then it would pass an integer token to the parser. The lexeme (*i.e.*, the matched text) of the token would be “99”. The lexer also removes all spaces, tabs, newlines, and comments from the program, simplifying the work of later compiler phases. Finally, the lexer finds all lexical errors, errors in the lexemes. For example, string constants cannot span multiple lines; a string that spans multiple lines is a lexical error. See a compiler textbook for more information on lexers [2, 17, 8, 14].

6.2 Files and Directories

All of the source code you will need to edit is within the lexer package. The particular files that you will need to modify depend on whether you’re using JLex or JavaCC to generate a lexer.

JLex. If you’re using JLex then you will need to modify the file **lexer.jlex** (and only the file **lexer.jlex**). This file contains an incomplete specification of the lexer, which you will need to complete. See the JLex manual [7] for more information about building JLex specification files.

There is also a (working) file called **Token.java** that contains a class for representing tokens. The lexer will need to pass a **Token** object to the parser, for each recognized token. See the API

html documentation (click on class **Token**) for how to construct a **Token** object. It will require knowing the identifier of the token, which can be found in the class **TokenIds** in the parser class.

Tokens should still be sent to the parser even when errors are discovered. There is a special token identifier (**LEX_ERROR**) that indicates a lexical error occurred.

JavaCC. If you're using JavaCC then you will need to modify the file **lexer.jj** (and only the file **lexer.jj**). This file contains an incomplete specification of the lexer, which you will need to complete. See the JavaCC manual [6] for more information about building JavaCC specification files.

6.3 Lexer Details

Your lexer must be able to handle the following types of tokens:

- Special symbols (*e.g.*, '{', '}', ';', *etc.*)
- Keywords (*e.g.*, **class**, **while**, *etc.*)
- Identifiers (*e.g.*, **foo**, **Foo**, *etc.*)

Identifiers are any non-keyword that starts with an uppercase or lowercase letter and is followed by a sequence of letters (upper or lowercase), digits, and '_'.

- Integer constants (*e.g.*, 9, 32, *etc.*)

Integer constants are any sequence of digits (with possibly leading zeros) that is between 0 and $(2^{31} - 1)$. Note: other integer constant forms such as hexadecimal are not supported.

- Boolean constants (**true** or **false**)

- String constants (*e.g.*, "abc")

String constants start and end with double quotes. They may contain the following special symbols: \n (newline), \t (tab), \" (double quote), \\ (backslash), and \f (form feed). A string constant cannot exceed 5000 characters and cannot span multiple lines.

Your lexer should remove all spaces, tabs, and newlines, as well as all comments. Bantam supports both multi-line comments (starting with **/*** and ending with **/***) and single line comments (starting with **//** and going until the end of the line).

Your lexer should identify the following lexical errors:

- Unterminated multi-line comments
- Unterminated string constants
- String constants spanning multiple lines
- String constants that are too long
- Unsupported escape characters within a string (*e.g.*, ‘\0’)
- Integer constants that are too large
- Unsupported characters (*e.g.*, ‘?’, ‘@’) that are not within a comment or string constant

6.4 Testing the Lexer

You can test your lexer in one of two ways. First, you can supply the “-sl” flag to the Bantam compiler, which stops compilation after lexical analysis and prints the scanned tokens. You can compare the printed tokens with the printed tokens from the reference compiler. If they are different then this probably indicates an error in your lexer. A second way to test your lexer is to use the provided class files for performing syntactic and semantic analysis, and code generation, and fully compile some programs. If this produces incorrect assembly code, then this probably indicates an error in your lexer.

As part of the assignment you should submit a Bantam source program that tests your lexer (**Test.btm**). It should contain as many kind of lexical errors as you can code in a single program. It should also contain each type of token.

Important: take testing seriously in this project and all of the remaining projects. It is a critical part of the software design process. There is a strong correlation between students with working compilers and students who thoroughly test their code.

6.5 Last Words

Remember to start early on this project and all the remaining projects. It will take a significant amount of time (more than you probably think) and the sooner you finish the more amount of time you can spend on testing your lexer. Good luck and remember to have fun!



Figure 6: Bantam Java compiler phases. The parser phase is shaded.

7 Project 2: Building a Parser

In the second project, you will build a parser using an automatic parser generator. You will need to complete the code within the parser package of your compiler source code. Make sure that you have read Section 3 on the Bantam Java programming language and Section 4 on the Bantam Java compiler before starting this project.

7.1 Overview

The parser is the phase of the compiler that checks that the program is syntactically correct and, in addition, builds an internal representation of the program. Figure 6 shows the parser phase (shaded) along with the other compiler phases.

The parser uses the lexer to build tokens from streams of characters (see Section 6). It then checks that these tokens form a syntactically correct sentence, *i.e.*, a correct program. For example, if a Bantam program includes a class without an end brace (‘}’) the parser should catch this error, print an appropriate error message, and stop compilation after parsing is complete.

As the parser is checking correctness, it builds an intermediate representation of the program. In the Bantam Java compiler, the program is represented using an abstract syntax tree, or AST. An AST has a node for every type of construct in the language (*e.g.*, class, if statement, multiplication expression, int constant, *etc.*). These nodes may contain sub-nodes depending on the particular construct. For example, a class AST node contains a list of member definitions (methods and fields). See Section 4 for the details of the AST. If the program is syntactically correct then the AST representing the source program is passed on to later phases of the compiler (*e.g.*, semantic analyzer, code generator). For more information on parsers and ASTs see a compiler textbook [2, 17, 8, 14].

7.2 Files and Directories

All of the source code you will need to edit is within the parser package. The particular files that you will need to modify depend on whether you're using Java Cup or JavaCC to generate a parser.

Java Cup. If you're using Java Cup then you will need to modify the file **parser.cup** (and only the file `parser.cup`). This file contains an incomplete specification of the parser, which you will need to complete. See the Java Cup manual [12] for more information about building Java Cup specification files.

JavaCC. If you're using JavaCC then you will need to modify the file **parser.jj** (and only the file `parser.jj`). This file contains an incomplete specification of the parser, which you will need to complete. See the JavaCC manual [6] for more information about building JavaCC specification files.

7.3 Parser Details

Figure 7 shows a specification of the Bantam Java syntax. The specification is similar to a context-free grammar, which is described in detail in any compiler textbook [2, 17, 8, 14]. This specification shows how we can generate any syntactically-correct program starting from the non-terminal *Program*. Note that the terminals *identifier*, *int_const*, *boolean_const*, and *string_const* are lexical tokens for identifiers (*e.g.*, variable names, class names), int constants (*e.g.*, 1, 29), boolean constants (*e.g.*, true, false), and string constants (*e.g.*, "abc", "def").

Parser generators use context-free grammars to automatically construct a parser that can recognize syntactically-correct programs. You will write a specification for the context-free grammar in either the Java Cup or JavaCC format (depending on which parser generator you are using). You will then run the parser generator and build your Bantam parser by building the source (see Section 2 for details on building the source). Once you have a working specification, your parser should recognize any program that can be generated from this specification. Any program that cannot be generated from this specification is syntactically incorrect. For these programs, your parser should halt compilation and print out appropriate error messages.

There are a few differences between the specification in Figure 7 and a context-free grammar. First, [and] are used to denote an optional sequence of terminals and non-terminals within the righthand side of some production. Second, * and + as superscripts are used to denote zero

```

Program → Class+
Class → class identifier [ extends identifier ] { Member* }
Member → Method | Field
Method → identifier identifier ( Formal* ) { Stmt* RetnStmt }
Field → identifier identifier [ = Expr ] ;
Formal → identifier identifier
Stmt → ExprStmt | DeclStmt | IfStmt | WhileStmt | BlockStmt
RetnStmt → return Expr ;
ExprStmt → Expr ;
DeclStmt → identifier identifier = Expr ;
IfStmt → if ( Expr ) Stmt [ else Stmt ]
WhileStmt → while ( Expr ) Stmt
BlockStmt → { Stmt* }
Expr → AssignExpr | DispatchExpr | NewExpr | InstanceofExpr |
      CastExpr | BinaryExpr | UnaryExpr | ConstExpr | VarExpr
AssignExpr → VarExpr = Expr
DispatchExpr → [ Expr . ] identifier ( Expr* )
NewExpr → new identifier ( )
InstanceofExpr → Expr instanceof identifier
CastExpr → ( identifier ) ( Expr )
BinaryExpr → BinaryArithExpr | BinaryCompExpr | BinaryLogicExpr
UnaryExpr → UnaryNegExpr | UnaryNotExpr
ConstExpr → int_const | boolean_const | string_const
BinaryArithExpr → Expr + Expr | Expr - Expr | Expr * Expr |
                  Expr / Expr | Expr % Expr
BinaryCompExpr → Expr == Expr | Expr != Expr | Expr < Expr |
                  Expr <= Expr | Expr > Expr | Expr >= Expr
BinaryLogicExpr → Expr && Expr | Expr || Expr
UnaryNegExpr → - Expr
UnaryNotExpr → ! Expr
VarExpr → [ identifier . ] identifier

```

Figure 7: Bantam Java syntax specification. Non-terminals start with capital letters and terminals start with lowercase letters or a non-alphabetic character. *Program* is the starting non-terminal.

or more occurrences or one or more occurrences, respectively. Finally, a comma as a subscript denotes comma-separated lists. A comma must appear between any two elements, but does not appear after the last element (if there are at least two elements – no comma is used for one or zero elements). You will need to figure out how to encode these into an unambiguous context-free grammar, which the parser generator will use to automatically construct the Bantam parser.

Your parser generator (Java Cup or JavaCC) also allows you to execute programmer-specified Java code, called *actions*, whenever a particular production is applied. You will use this feature to perform syntax-directed translation, in this case, translating the Bantam source program into an

abstract syntax tree. See the Java Cup or JavaCC manual for the details on how to define actions.

7.4 Error Handling

Error handling is an important component of the compiler. The compiler must identify errors and print meaningful error messages so that the Bantam programmer can fix any errors. At a bare minimum, if your parser encounters a token that it was not expecting, it should print “Unexpected token” along with the filename and line number where the error occurred. For extra credit, identify specific errors such as a missing return statement, a nested class or method, or an unterminated class.

Your parser must halt compilation if at least one syntactic error is encountered, *i.e.*, the semantic analyzer and code generator should not run. For syntactic analysis, it is enough to stop parsing after one error is encountered (this will not be acceptable for semantic analysis). Extra credit, however, will be awarded for parsers that can catch multiple syntactic errors during a single compilation. You will need to be able to recover from an error in order to implement this. See the Java Cup or JavaCC manual for details on implementing error recovery.

An **ErrorHandler** class is provided in the util package for printing and managing errors. See the API html documentation (click on class **ErrorHandler**) for details on using it.

7.5 Testing the Parser

You can test your parser in one of two ways. First, you can supply the “-sp” flag to the Bantam compiler, which stops compilation after syntactic analysis and prints the parsed program as a Bantam source program. You can compare the parsed program with the parsed program from the reference compiler. If they are different then this probably indicates an error in your parser. A second way to test your parser is to use the provided class files for performing lexical and semantic analysis, and code generation, and fully compile some programs. If this produces incorrect assembly code, then this probably indicates an error in your parser.

As part of the assignment you should submit two Bantam source programs that test your parser (**BadTest.btm** and **GoodTest.btm**). The first, **BadTest.btm**, should contain as many kinds of syntactic errors as you can code in a single program. The second, **GoodTest.btm**, should contain a working Bantam source program that tests all of the Bantam syntax rules.

Important: take testing seriously in this project. A parser is more complicated and thus more error-prone than a lexer, so be prepared to spend even more time debugging than in the previous project. Remember, testing is a critical part of the software design process. There is a strong correlation between students with working compilers and students who thoroughly test their code.

7.6 Last Words

Remember to start early on this project. It will take a significant amount of time (more time than the previous project) and the sooner you finish the more time you can spend on testing your parser. Good luck and remember to have fun!



Figure 8: Bantam Java compiler phases. The semantic analysis phase is shaded.

8 Project 3: Building a Semantic Analyzer

In the third project, you will build a semantic analyzer. You will need to complete the code within the `semant` package of your compiler source code. Make sure that you have read Section 3 on the Bantam Java programming language and Section 4 on the Bantam Java compiler before starting this project.

8.1 Overview

The semantic analyzer is the phase of the compiler that checks that the program is semantically correct. Your semantic analyzer will also annotate the intermediate representation with type information needed by the code generator, and will build a data structure for representing the class hierarchy tree, which is important for both semantic analysis and code generation. Figure 8 shows the semantic analysis phase (shaded) along with the other compiler phases.

The semantic analyzer is passed the abstract syntax tree or AST (the intermediate representation) from the parser (see Section 7), assuming the program is syntactically correct. The semantic analyzer traverses the AST and checks that the program is semantically sound. For example, the semantic analyzer must verify that the operands in a multiplication expression both have type `int` (*i.e.*, it must *type check* the expression). If this is not the case, then the semantic analyzer should catch this error, print an appropriate error message, and stop compilation after semantic analysis is complete.

Therefore, in order to check that the program is semantically correct, your semantic analyzer will need to determine the type of each expression in the AST. The code generator also needs this information in order to generate correct assembly code. So it is the responsibility of the semantic analyzer to annotate each expression node with its discovered type.

Because Bantam Java supports inheritance, to check that the program is semantically correct, your semantic analyzer will need a data structure for representing the class hierarchy tree (see Section 3 for a description of the class hierarchy tree). For example, if an expression with type **A** (an object type) is used at a place in the Bantam source program where type **B** (also an object type)

is required, this code is legal only if **A** is a subclass of **B**. With a data structure for representing the class hierarchy tree, this property can be verified.

For more information on the theory of semantic analysis see a compiler textbook [2, 17, 8, 14].

8.2 Files and Directories

All of the source code you will need to edit is within the `semant` package. In particular, you will need to complete the file, `SemanticAnalyzer.java`, which contains some skeleton code for performing semantic analysis. You may also find it helpful to add other Java files to this package. For example, you might write a separate class (in a separate Java file) for traversing the AST and performing type checking (*i.e.*, checking that types are used correctly within the program).

In addition, you will find many of the files in the `util` and `visitor` packages useful, although you should not need to modify any of them. For example, in the `util` package, the file `ClassTreeNode.java` contains a class for representing nodes in the class hierarchy tree, the file `SymbolTable.java` contains a class for representing a symbol table, and the file `ErrorHandler.java` contains a class for managing and printing errors. In the `visitor` package, the file, `Visitor.java` contains an abstract class, which can be extended in order to traverse the AST using the visitor design pattern [11].

8.3 Semantic Analyzer Details

Your semantic analyzer will need to perform the following four steps:

1. Build a data structure representing the class hierarchy tree and check that it is well-formed (*e.g.*, there are no cycles, no repeated classes, *etc.*). You should use the class **ClassTreeNode** in the `util` package for representing each node in the tree. Each node contains some meta-information about the class (*e.g.*, a link to the parent and children nodes) and also the AST node for that particular class. The code generator takes the root of this tree (*i.e.*, the **Object** node) as input.
2. Build class environments. In order to type check the program (in the final step) you will need to know what methods and fields are defined in a particular class. For example, if your semantic analyzer encounters `x.foo()` during type checking, where `x` has type **X**, your semantic analyzer will need to look up `foo` in class **X** and verify that it exists and it is used correctly.

So in this step (2), we add all class members (inherited or otherwise) to the class's symbol table (using the symbol tables is discussed below). In addition, your semantic analyzer will need to check that methods and fields are defined correctly (*e.g.*, there are no methods or fields defined with the same name).

3. Check that the **Main** class is defined and contains a **main()** method, which has no parameters and no return value. Note: the **Main** class can extend another class and the **main()** method could be inherited.
4. Type check each field initialization expression and each method, annotating expressions as we type check. Anywhere a particular type is expected, you will need to verify that that type is used or a subtype is used (if the type is an object type). You will also annotate all expression nodes with their type, which is needed by the code generator.

Read Section 3 closely for details on the semantics of the Bantam Java language. For instance, Section 3 specifies the correct operand types for each particular expression. It also specifies the return type for each particular expression. In addition, Section 3 specifies the rules for method calls and for redefining methods and variables in an extended class.

Data structures. Your semantic analyzer will need to build and use two important data structures: a class hierarchy tree and a symbol tree. Both of these are provided in the util package.

The class **ClassTreeNode** is used for representing a node in the class hierarchy tree. After (or while) checking that the class hierarchy tree is well formed, your semantic analyzer should then build a representation of the tree using **ClassTreeNode**. Each node in this tree contains all the information pertinent to that corresponding class including the name, the links to the parent and children class tree nodes, and a link to the AST node representing that class. The root of this tree, which is the node representing the built-in class **Object**, is passed to the code generator. See Section 4.2 or the html API for more information on how to use this class.

The class **SymbolTable** is used for representing a symbol tree. Each class tree node has two symbol tables: one for variables and one for methods. Variables and methods can be added to these symbol tables in order to later type check the program. For example, during type checking, if your semantic analyzer encounters an assignment to a variable **x**, it can look up **x** in the variable symbol table to make sure that it exists and that the type matches the type on the righthand side of

the assignment. A similar strategy can be used when your semantic analyzer encounters a method call.

These symbol tables support inheritance; each symbol table has a link to the corresponding symbol table in the parent class. When looking up a variable or method, first the current class's symbol table is searched, and then if the symbol is not found, the parent class's symbol table is searched. This process continues until the top of the class hierarchy tree is reached.

The symbol table also supports nested scopes. For example, a block statement defines a new nested name space (nested within the current name space). When searching for symbols in the symbol table, your semantic analyzer can search either in the current scope (called a *lookup*) or in any scope (called a *peek*). See Section 4.2 or the html API for more information on how to use this class.

Visitor pattern. Your semantic analyzer will need to traverse the abstract syntax tree (AST) in order to perform semantic analysis. While you should not walk the AST needlessly, do not feel as though you need to optimize the total number of traversals (perhaps, by combining multiple traversals into one). Such optimizations will not reduce the compile time by all that much in most cases, and the performance of the compiler is generally not critical (as opposed to the performance of the compiled code). One approach might be to traverse the AST once and build the symbol tables for each class (bullet two in the four steps described above) and a second time to type check each class (bullet four in the four steps described above).

As described in Section 4.2, the best approach for traversing an AST from a software engineering perspective is to use the visitor design pattern. This approach allows us to keep the AST functionality separate from the semantic analysis functionality. It also makes it easier to modify and maintain the semantic analyzer. A change in the semantic analyzer has no impact on the AST, which is important since there are 51 classes representing each particular type of AST node. To write a visitor class, you will write a class that extends the abstract class **Visitor**. This class will override each **visit** method with a method that traverses the AST and performs the additional functionality (*e.g.*, type checking). For an example, look at the **PrintVisitor** class in the visitor package, which traverses the AST and prints each node. For a more detailed description of the visitor design pattern, see Section 4.2.

8.4 Error Handling

Like in the parser, error handling is an important component of the semantic analyzer. The compiler must identify errors and print meaningful error messages so that the Bantam programmer can fix any errors. Unlike with parsing, error identification in the semantic analyzer should be more precise and more thorough. Your semantic analyzer should find all semantic errors within the program, assuming that one error does not hide another error. For example, imagine a programmer attempts to call a method **foo**, which takes an **int** as a parameter, and mistakenly mistypes the method name and forgets to pass any parameter in the call. Only the first error can be discovered in this case, the second error will only be discovered after the programmer fixes the first error. But excluding errors hidden by other errors, your semantic analyzer should find all semantic errors within the program.

For each discovered error, you should print a meaningful error message and also print the file-name and line number where the error occurred. For semantic analysis, unlike syntactic analysis, it is usually much easier to give a precise error message. Run the reference compiler (discussed underneath the heading “Testing the Semantic Analyzer”) to see some examples of good error messages.

An **ErrorHandler** class is provided in the util package for printing and managing errors. See the API html documentation (click on class **ErrorHandler**) for details on using it.

8.5 Testing the Semantic Analyzer

You can test your semantic analyzer in one of two ways. First, you can supply the “-ss” flag to the Bantam compiler, which stops compilation after semantic analysis and prints the intermediate representation as a Bantam source program. Annotations (*i.e.*, expression types) are provided within comments in the source program. You can compare this output with the corresponding output from the reference compiler. If the output is different then this probably indicates an error in your semantic analyzer. A second way to test your semantic analyzer is to use the provided class files for performing lexical and syntactic analysis, and code generation, and fully compile some programs. If this produces incorrect assembly code, then this probably indicates an error in your semantic analyzer.

Important: take testing seriously in this project. A semantic analyzer is more complicated and thus more error-prone than a parser or a lexer, so be prepared to spend even more time debugging than in the previous two projects. Remember, testing is a critical part of the software design

process. There is a strong correlation between students with working compilers and students who thoroughly test their code.

As part of the assignment you should submit two Bantam source programs that test your semantic analyzer (**BadTest.btm** and **GoodTest.btm**). The first, **BadTest.btm**, should contain as many kinds of semantic errors as you can code in a single program. The second, **GoodTest.btm**, should contain a working Bantam source program that tests all (or most) of the Bantam semantic rules described in Section 3.

8.6 Last Words

Remember to start early on this project. It will take a significant amount of time (more time than the previous two projects) and the sooner you finish the more amount of time you can spend on testing your semantic analyzer. Good luck and remember to have fun!

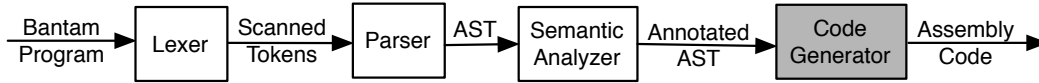


Figure 9: Bantam Java compiler phases. The code generation phase is shaded.

9 Project 4: Building a Code Generator

In the fourth and final project, you will build a code generator. If you are targeting the Mips architecture, you will need to complete the code within the `codegenmips` package of your compiler source code. Otherwise, if you are targeting x86, you will need to complete the code within the `codegenx86` package of your compiler source code. Before starting this project, make sure that you have read Section 3 on the Bantam Java programming language, Section 4 on the Bantam Java compiler, and most importantly, Section 5 on the Bantam Java runtime system.

9.1 Overview

The code generator is the final phase of the compiler that takes as input the internal representation of the program and produces assembly code (other compilers may target machine code or a virtual machine, but the Bantam Java compiler targets assembly code). Figure 9 shows the code generation phase (shaded) along with the other compiler phases.

The code generator is passed the root node of the class hierarchy tree from the semantic analyzer. From the root node, we can of course reach any other class tree node. Each node contains meta-information about the class as well as a link to the AST node representing that particular class. Any expression AST nodes embedded within the class AST node is annotated with its type, which was done by the semantic analyzer. The code generator can assume that the program it is passed is a legal, Bantam program. The code generator traverses the class hierarchy tree and AST nodes (perhaps, multiple times) and generates assembly code.

For more information on the theory of code generation see a compiler textbook [2, 17, 8, 14].

9.2 Files and Directories

All of the source code you will need to edit is within either the `codegenmips` package or `codegenx86` package (depending on the particular target you are using). In particular, you will need to complete the file, `MipsCodeGenerator.java` or `X86CodeGenerator.java` (again, depending on the

target), which contains some skeleton code for performing code generation. You may also find it helpful to add other Java files to this package. For example, you might write a separate class (in a separate Java file) for traversing each AST node and generating assembly code.

In addition, you will find many of the files in the `util` and `visitor` packages useful, although you should not need to modify any of them. For example, in the `util` package, the file `ClassTreeNode.java` contains a class for representing nodes in the class hierarchy tree and the file `SymbolTable.java` contains a class for representing a symbol table. In the `visitor` package, the file, `Visitor.java` contains an abstract class, which can be extended in order to traverse the AST using the visitor design pattern [11].

9.3 Code Generator Details

The code generator will need to work closely with the provided runtime system, which is discussed in Section 5. In particular, the code generator will need to use the same calling conventions as the runtime system and to layout data structures in memory as the runtime system expects. Many of the details are discussed in Section 5.

Your code generator will need to generate several different data structures and subroutines. The data structures must appear within the data section, *i.e.*, they must follow the `.data` directive. The subroutines must appear within the text section, *i.e.*, they must follow the `.text` directive. A few of these are already implemented within the compiler, but most are not. In particular, your code generator will need to generate the following:

- **Globals.** Several of the data structures and subroutines discussed below must be made available to the runtime system. The data structures that must be made global include `gc_flag`, `class_name_table`, `Main_template`, `String_template`, and `String_dispatch_table`. The subroutines that must be made global include `Main_init` and `Main.main`.
- **Garbage collection.** Your code generator will need to set a flag called `gc_flag` to 1 or 0 depending on whether garbage collection is enabled or disabled, respectively. For example, if garbage collection is disabled, then the following code should appear in the data section:

```
gc_flag:    .word 0
```

- **String constants.** Your code generator will need to scan the program to find all string constants (*e.g.*, “abc”). For each string constant, you should generate a string object within the data section of the assembly program. Give each string constant a unique name. For example, you could use **String_const_<id>** to name each string constant where <id> is replaced with a unique numeric identifier. When that string constant is used in the program, you can use the address of the string object (which in assembly is simply the name). You will probably also need a data structure for mapping a string constant (*e.g.*, “abc”) to its string object reference name (*e.g.*, **String_const_5**). As discussed in Section 5, you will also need to generate string objects for each class name (built-in or user-defined) for use in error reporting.
- **Class name table.** Your code generator will need to build a class name table (called **class_name_table**) that contains pointers to the strings representing the class names. This table is used by the runtime system for error reporting when a runtime error occurs. It is ordered by class identifier. For example, if the class **Object** has identifier 0, then a pointer to a string representing “Object” should be contained in the first entry.
- **Object templates.** Your code generator will need to generate object templates (*e.g.*, **TextIO_template**) for each built-in and user-defined class. See Section 5 for details on how to construct these templates.
- **Dispatch tables.** Your code generator will need to generate dispatch tables for each built-in and user-defined class (*e.g.*, **String_dispatch_table**). See Section 5 for details on how to construct these templates.
- **Initialization subroutines.** Your code generator will need to generate initialization subroutines for each built-in and user-defined class (*e.g.*, **Main_init**). These should be named using the following format:

<class name>_init

These subroutines are needed during object construction. Objects are created by cloning (with `Object.clone`) the appropriate template. The template contains default values for each field in the object. The initialization subroutine evaluates each field’s initialization expression and uses the resulting value to set the corresponding field. Fields are initialized in the

order that they appear within the class. Inherited fields are initialized before non-inherited fields.

- **User-defined methods.** Finally, your code generator will need to generate code for each user-defined method. It will not need to generate code for built-in methods since these are defined within the runtime system. Each method should be named using the following format:

`<class name>.<method name>`

For example, the method **main** within the class **Main** would be named **Main.main**. These methods should use the calling conventions discussed in Section 5.

Your code generator will probably need to perform several traverses over the internal representation of the program. For example, you could traverse the program once to find string constants, a second time to generate initialization subroutines, and a third time to generate user-defined methods. Create visitor classes to traverse the program as in project 3.

As in project 3, your code generator will need to use the symbol table to maintain information about various aspects of the program. For example, the symbol table can be used to maintain the location of each variable in memory or in register. To represent the location, you can use the **Location** class defined within the **util** package. See the API for details.

Do not worry about generating optimal code or optimizing the running time of your code generator. Building a working code generator will be very challenging without worrying about optimization.

For extra credit, add some optimizations to improve the performance of the generated code. For example, you could perform some data flow analysis (*e.g.*, available expressions) and use this analysis to perform an appropriate optimization (*e.g.*, common-subexpression elimination). The amount of extra credit will vary greatly based on the amount of work that you do. Under no circumstances should you attempt the extra credit until your code generator is working!

9.4 Testing the Code Generator

Unlike the previous phases, there is only one way to test your code generator. You can use the provided classes for performing lexical, syntactic, and semantic analysis, and fully compile some

programs. You can then try comparing the generated assembly program to the generated assembly program from the reference compiler. However, the output from your code generator could be different from the reference output, and still be correct. If the two generated programs are different, then you can run them and see if the programs have the same behavior. If they do not have the same behavior, then this probably indicates an error in your code generator.

It might be worth using the same format for your generated assembly program as the reference compiler. If you are able to achieve this, then you will be able to simply compare the two outputted assembly files, which is much easier than comparing the behavior of the programs when they are run. However, if you find this too difficult to do, then you may want to simply compare the compiled program's behavior when executed.

As part of the assignment you should submit a Bantam source program that tests your code generator (**Test.btm**). The file should contain a working Bantam source program that uses as much of the Bantam language as you can code within a single program.

Important: take testing seriously in this project. A code generator is by far the most complicated phase of the compiler, so be prepared to spend even more time debugging than in the previous projects. Remember, testing is a critical part of the software design process. There is a strong correlation between students with working compilers and students who thoroughly test their code.

9.5 Last Words

Remember to start early on this project. It will take a significant amount of time (more time than the previous projects) and the sooner you finish the more amount of time you can spend on testing your code generator. Good luck and remember to have fun!

10 Closing Remarks

Hopefully if you have reached this point then you have successfully implemented your very own compiler. If so, congratulations! There are few accomplishments in an undergraduate computer science program as significant as implementing a compiler. Nice work.

We hope that you have enjoyed the Bantam Java course projects and found this manual helpful and informative. Please let us know if you found any bugs or errors along the way, in either the Bantam Java toolset or in this manual. In addition, please contact us if you have questions about any part of this work or if you have requests for extensions. Finally, we welcome contributions from others, so please let us know if you would like to contribute in some way. To contact us you can email the first author at *corliss@hws.edu* or you can visit *http://www.bantamjava.com*, which has a link to an online bulletin board for submitting questions, comments, or requests.

Thanks,

Marc Corliss & E Christopher Lewis

References

- [1] GCC, the GNU compiler collection. URL: <http://gcc.gnu.org>.
- [2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley, 2nd edition, 2007.
- [3] Alexander Aiken. Cool: A portable project for teaching compiler construction. *SIGPLAN Notices*, 31(7):19–24, 1996.
- [4] Andrew Appel. *Modern Compiler Implementation in Java*. Cambridge, 1st edition, 1998.
- [5] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison-Wesley, 4th edition, 2005.
- [6] Elliot Berk. JavaCC documentation. URL: <https://javacc.dev.java.net/doc/docindex.html>.
- [7] Elliot Berk. *JLex: A lexical analyzer generator for Java*, 1997. URL: <http://www.cs.princeton.edu/~appel/modern/java/JLex/current/manual.html>.
- [8] Keith D. Cooper and Linda Torczon. *Engineering a Compiler*. Morgan Kaufman, 2004.
- [9] Marc L. Corliss and E Christopher Lewis. Bantam: A customizable, java-based, classroom compiler. In *Proc. of the 39th SIGCSE Tech. Symp. on Computer Science Education*, Mar. 2008.
- [10] David Eck. *Introduction to Programming using Java - Part 1*. Creative Commons, 5th edition, 2006.
- [11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [12] Scott Hudson. *Cup User's Manual*, 1999. URL: <http://www.cs.princeton.edu/~appel/modern/java/CUP/manual.html>.
- [13] Jim Larus. Spim s20: A mips r2000 simulator. Technical Report TR-966, Computer Sciences Department, University of Wisconsin-Madison, Sep. 1990.
- [14] Thomas W. Parsons. *An Introduction to Compiler Construction*. W H Freeman and Co, 1992.

- [15] Sun Microsystems, Inc. *javadoc - The Java API Documentation Generator*, 2002. URL: <http://java.sun.com/j2se/1.5.0/docs/tooldocs/solaris/javadoc.html>.
- [16] Niklaus Wirth. *Compiler Construction*. Addison-Wesley, 1996.
- [17] Andrew Appel with Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge, 2nd edition, 2002.