RICE UNIVERSITY

Adding Support for Language Levels to DrJava

by

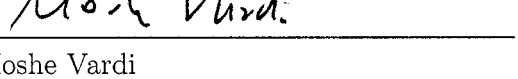
James I. Hsia

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

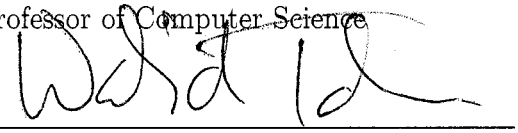
MASTER OF SCIENCE

Approved, Thesis Committee:

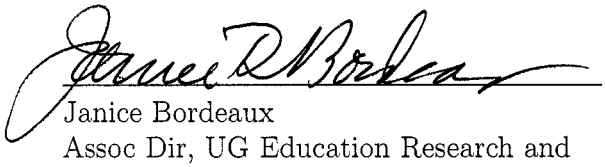
Robert Cartwright , Chair
Professor of Computer Science



Moshe Vardi
Professor of Computer Science



Walid Taha
Assistant Professor of Computer Science



Janice Bordeaux
Assoc Dir, UG Education Research and
Assessment

Houston, Texas

June, 2004

ABSTRACT

Adding Support for Language Levels to DrJava

by

James I. Hsia

This thesis describes the design and implementation of *language levels*, a pedagogic tool that helps students progressively learn object-oriented programming concepts in discrete stages. While Java is widely used in academia, its complex syntax and large array of constructs are difficult for beginning students to learn. The language levels facility supports a hierarchy of progressively sophisticated subsets of the language. This progression minimizes the clerical burden involved in learning to write Java programs and reinforces the specific abstractions taught at each stage of Rice introductory programming curriculum. In addition, the hierarchy of language levels reduces code clutter by automatically generating routine methods.

This language levels facility has been implemented as an extension of DrJava, a pedagogic programming environment for Java developed at Rice University. We anticipate that this extension will enable more students to learn Java and master the principles of object-oriented programming.

Acknowledgments

I would like to thank my adviser, Professor Robert Cartwright, for inspiring and guiding my research. His passion for his work exemplifies what I would like to see in my own life.

Elspeth Simpson has tirelessly worked with me on the implementation of language levels. Without her diligence and level-headedness, this project would not have gotten far.

I also thank Charlie Reis for demonstrating by example how to be a great program manager.

Finally, I am also grateful to the students in Comp312 for their support and feedback.

Contents

List of Tables

Chapter 1
Introduction

This thesis describes the design and implementation of a hierarchy of progressively more sophisticated *language levels*, each consisting of a subset of the Java language. While Java is widely used in introductory programming courses, its complex syntax and large array of constructs make it difficult for beginning students to learn. A hierarchy of language levels helps beginning students by restricting the data models and computations that can be expressed as well as the range of constructs available to implement these models. Our language levels facility partitions the Java language into three levels: Elementary, Intermediate, and Advanced, in addition to the full language. Each successive level in the hierarchy embodies a richer, more complex collection of abstractions for defining data and describing computations over that data.

1.1 Motivation

In a conventional introductory programming course, the canonical first program simply prints out the text, "Hello World." Unfortunately, even writing this trivial program in Java involves many different constructs of the language, some of which are quite advanced. To be executable from the command line, a Java class must contain the method header:

```
public static void main (String[] argv)
```

This line of code relies on the concepts of static methods, visibility modifiers, return types, the "main method" convention for command line execution, formal parameters, and arrays! In most introductory Java curricula, Java programming is thus introduced as a collection of cryptic, unrelated ideas [19]. Instructors must either explain each of these concepts before students write any code or require their students to copy-and-paste code they do not understand, both of which are poor choices [30]. This example clearly demonstrates the need for a pedagogic programming environment supporting a simpler linguistic interface than the standard command line interface designed for experienced programmers. Such an interface would enable introductory curricula to introduce Java language constructs and the associated data models and computations in smaller, more easily digested units. Much more time could be devoted to teaching object-oriented programming principles since less time is spent struggling with the complexities of the full Java language.

Java's complexity can be ameliorated by using pedagogic *integrated development environments* (IDEs) like DrJava [1] and BlueJ [23] that provide more convenient computational interfaces than a command line. These pedagogic IDEs allow arbitrary methods to be called from an interactions window, bypassing the command line interface.

Pedagogic IDEs, however, still do not address the issue of Java's complex gram-

mar and large array of constructs. They support the entire Java language and only accept full Java class definitions. As a result, these environments explain malformed syntax in terms of the entire language using error messages that can refer to syntactic constructs foreign to beginners.

For example, if the closing brace is omitted from a class in a file containing several classes, any subsequent classes will be interpreted as inner classes; the compiler will report that a brace is missing at the *end of the program* after the closing brace for the last class. A pedagogic environment supporting a hierarchy of language levels solves this problem by reporting error diagnostics in terms of the current language subset that students understand. Syntax errors that mimic more advanced language constructs not in the subset are correctly flagged as errors rather than misinterpreted as more advanced constructs. In the missing brace example, beginning users would be alerted that a class definition cannot be nested inside another class (which our language levels prohibit at the Elementary and Intermediate levels). As a result of these level-specific error messages, students gain confidence in their understanding of each language level, encouraging them to master the computational power and corresponding syntax currently available.

While the optimal order for teaching object-oriented concepts is the subject of debate, we have had excellent results with the pedagogy developed by the Computer Science faculty here at Rice University [11]. We begin by teaching functional pro-

gramming over algebraic types like numbers, lists, and trees because it is a simple but surprisingly rich generalization of the familiar concepts of function definition and evaluation that students learn in arithmetic and algebra classes in grammar and secondary school. Next, we extend the domain of computation to include functions themselves by using them as arguments. Finally, mutation operations on data (reassignment to variables and fields) are introduced. Data mutation is postponed until fairly late in the first semester curriculum because it breaks many of the laws assumed in intuitive mathematical reasoning such as the "substitution of equals for equals."* In practice, many important computations in production programs are best formulated using immutable data [4].

For the past eight years, our introductory curriculum has mandated a semester of (mostly) functional programming in Scheme followed by a semester of object-oriented programming in Java. The first few weeks of the second semester course recast all of the design concepts from the first semster course in object-oriented terms using *design patterns* [16]. Recently, we have introduced an alternate first year sequence, taught entirely in Java, targeted at engineering and science majors outside of Computer Science, many of whom have little use for learning functional programming in Scheme. The new curriculum covers almost exactly the same progression of programming concepts as our original Scheme/Java curriculum.

*Many programming language researchers refer to the failure of these familiar algebraic laws as "loss of referential transparency."

The primary disadvantage of using Java instead of a functional language like Scheme to express simple computations over immutable data is that Java is very wordy and forces the programmer to write a great deal of "boilerplate" code that is implicit in the functional model of computation. Specifically, Java requires a "functional" program to explicitly define a data constructor, data accessors, a string representation function, and the equality operation (forcing redefinition of the corresponding hash code operation) for each form of algebraic data. *All of these operations are generated automatically in Scheme by a simple declaration of a data definition signature* (a define-struct declaration). In functional languages, only the declaration of the type name and the names and types of the fields for each form of algebraic data is necessary.* To emulate the same level of simplicity in Java, our Elementary and Intermediate language levels automatically generate these methods for users. Thus, our language levels are technically not *executable* subsets of the Java language; we must augment code written in these levels with generated code that implements the methods that are implicit in the functional programming paradigm. The resulting brevity of programs written at these first two levels is very close to that of Scheme and other functional languages.

The implementation of language levels presents a formidable task. Each level has

*Technically even the types of the fields can be omitted in Scheme, but we teach that they must always be specified as part of the documentation for the data definition. In statically typed functional languages like ML and Haskell, the compiler requires types for all fields.

its own grammatical rules, automatically generated code, and error diagnostics. As a result, each level requires a custom parser and type checker, effectively constituting a compiler without a code generator.

To facilitate code sharing, the implementation is partitioned into four phases. The first phase uses a coarse form of parsing shared across all language levels that scans a source file and converts it to an *abstract syntax tree* (AST). In the second phase, a level-specific syntax checker (implemented using the *visitor* design pattern) walks the AST, ensuring that it only contains permitted language constructs and building a symbol table recording the types for the members of each class in the AST. The third phase performs type-checking by walking the AST, confirming that syntactic components for each language construct have the proper types (based on the member types declared in symbol table). The final step in processing a given language level is to generate a legal Java program corresponding to the abbreviated code in the source file by augmenting this code with the "boilerplate" code dictated by the this language level. The generated Java source file is fed to the Java compiler for translation into Java bytecode.

Any software tool should be carefully constructed to guard against bugs or deficiencies in meeting its specifications. This precept is especially true when developing a programming environment for beginning programmers. Beginners are more likely to be confused when they encounter a bug in a programming environment because

they presume that tools work flawlessly and attribute any problems to their own code. In the context of supporting language levels, it is critically important that the error-checking performed by each language-level processor is consistent with the checking done by standard Java compilers.* Otherwise, a program that nominally conforms to the restrictions of a given language level may fail when it is subsequently compiled by the standard compiler for execution.

To ensure the robustness of the software we develop, the DrJava development team follows a programming methodology based on the central principles of eXtreme Programming (XP) [22]: *test-driven development, pair programming, continuous integration*, and *on-site customer involvement.* Following these tenets of XP increases the quality of the DrJava code base, provides extensibility, and ensures that knowledge of the code base is maintained in the face of high developer turnover.

The remainder of this thesis is organized as follows. Chapter 2 surveys other Java teaching tools. Chapter 3 provides our rationale for using DrJava and examines the integration process. Chapter 4 describes the design of each language level, while Chapter 5 discusses the implementation process. Chapter 6 covers the programming methodology used when writing language levels, and Chapter 7 outlines future work to be done on language levels.

*Language level checking must subsume the checking performed by the standard compiler.

Chapter 2
Related Work

2.1 Pedagogic IDE's

Since Java has become so popular in both education and industry, many tools have been developed to help novices learn how to program in Java. Conventional IDEs make the programming task simpler by integrating utilities such as code editors, compilers, and debuggers and by presenting program syntax and execution errors to users in an understandable format. Pedagogic IDEs go beyond this by catering to programmers who have little experience with Java or the obect-oriented (OO) approach to program design that Java was designed to support.

2.1.1 DrJava

DrJava [1] is a free, open-source, lightweight IDE written at Rice University by undergraduate and graduate students under the direction of Professor Robert Cartwright. DrJava's design follows three guiding precepts: *simplicity, interactivity,* and *an emphasis on textual program representation.* To achieve simplicity, DrJava supports a user interface consisting of only three panes:

1. a Documents Pane listing the open documents with the current document highlighted;

2. a Definitions Pane where the current document is displayed and available for

editing; and

3. an Interactions Pane where the user can evaluate arbitrary Java statements and expressions.

To prevent beginning programmers from being overwhelmed by the environment, DrJava keeps the the number of buttons and menu items displayed to a bare minimum.

The Interactions Pane transforms Java from a batch-oriented language to an interactive one. Users can dynamically create new instances of classes and invoke methods that are defined in one of the open documents or are part of the standard Java libraries. As a result, they can test various parts of their programs without having to define a new main method for each experiment, compile the enclosing file, and execute the enclosing class.

The third precept, an emphasis of textual program representation, is addressed through the use of correct syntax highlighting, brace matching, and uniform indenting. These features allow beginners to become proficient at editing Java code instead of manipulating it indirectly through a visualization medium such as UML notation. As a result, programmers using DrJava are not limited in their expressiveness by the constraints of visual program representations [28]. Nor are they misled by incorrect syntax highlighting.

2.1.2 BlueJ

BlueJ [23] is a free pedagogic IDE jointly developed by John Rosenberg of Monash University in Melbourne, Australia and by Michael Kölling of the University of Southern Denmark. BlueJ is designed to support teaching OO programming in Java to beginners. It emphasizes a visual rather than textual representation of programs, using class diagrams and an object workbench where programmers can interact with visual representations of objects using a graphical interface. BlueJ is extensible and is open to user submissions. The graphical perspective used in BlueJ helps beginners visualize class relationships, and the object workbench provides some of the interactive capabilities of DrJava. On the other hand, this interface does not scale well to larger programs. Interactions that involve forming complex expressions or accumulating many intermediate results are awkward to perform using the graphical interface. In addition, programmers must eventually learn to directly edit Java program text, which they largely bypass when using BlueJ.

2.1.3 Ready to Program

Ready to Program with Java Technology [18] is an IDE developed by Holt Software. It is designed to exhibit four properties: ease of use, ease of administration, low hardware requirements, and educator useful features. To this end, it uses a simple batch-oriented interface, features syntax highlighting and indenting, and supports a

user-defined library framework. It is packaged of as a stand-alone application including its own compiler and Java Virtual Machine. In contast to DrJava and BlueJ, it is available only on Windows and is marketed as a commercial IDE. The embedded JVM conforms to the Java SDK 1.2 specification. To support newer releases of Java, the user must download and install a newer Java JRE in a specific subfolder of the Ready to Program installation directory. Since the beginning of 2003, there does not appear to have been any active development on Ready to Program [34].

2.2 Teaching Languages

Since Java's complex grammar is a widely recognized problem in introductory programming courses, there have been several major efforts to simplify the process of learning the Java language. One approach is to develop a pedagogic language similar to Java with a shallower learning curve. The primary disadvantage of this approach is that students must subsequently learn the actual Java language, which can be a time-consuming process. In addition, starting the programming curriculum with a "toy" language may diminish students' motivation and enthusiasm for learning OO programming.

2.2.1 MiniJava

MiniJava [29] is a pedagogic language consisting of a restricted subset of Java developed by Eric Roberts at Stanford University. It was designed to make Java's

complex syntax more accessible to novices. MiniJava restricts the use of inner classes, the do-while statement, the continue statement, and requires users to insert a break statement after each case in a switch statement. It also simplifies I/O and graphics by providing custom classes. In addition, the language supports autoboxing and unboxing (subsequently added to Java in version 1.5), eliminating most of the distinctions between primitive values and objects. Unfortunately, the development of MiniJava appears to have been abandoned before a full implementation was produced.

2.2.2 Karel the Robot

Karel the Robot [25] is a simple procedural language for controlling a turtle robot (Karel) designed to teach the fundamental concepts and skills of programming. Karel was originally developed by Richard Pattis at Carnegie Mellon University as a library for Pascal and has subsequently been ported to C++ and Java. The Java version has been developed by Joseph Bergin at Pace University. Karel consists of a library of procedures/methods that command a robot to navigate through an imaginary grid-like world performing a few simple interactions with that world. For example, users can call the TurnOn() method to begin a sequence of commands and Move() to command Karel to go forward a square. Karel emphasizes procedural logic and structure over numerical calculation and uses a simple, entertaining computational model (controlling a robot) to engage students and help them learn the fundamentals of procedural programming including control flow, syntactic nesting, loops, and procedures. By

design, Karel the Robot teaches procedural rather than OO programming concepts. In the Java version, students can begin to learn some object-oriented programming principles but only after they have learned to program in a procedural style.

2.2.3 Kenya

Kenya [7] is a teaching language designed by Robert Chatley, a Ph.D. student at Imperial College London. Kenya uses a translator to generate corresponding Java code. Because of the difficulty involved in learning Java, Kenya was created as a simpler language loosely based on Java. There are three goals underlying the design of Kenya: syntactic simplicity, compatibility with Java syntax where appropriate, and sufficient expressiveness for solving typical introductory programming problems. The resulting language is an amalgam of Java and Basic and does not foster an OO perspective on programming. As a result, OO programming concepts cannot be taught in Kenya and must be deferred until after the transition to Java. In addition, the transition to Java syntax is difficult because programming conventions taken from Basic must be unlearned in the process.

2.2.4 JJ

JJ [27] is a programming language developed by John Motil and associates at California State University, Northridge and has a supporting IDE. JJ has a syntax based on Java that is designed to be easier to learn than Java. JJ uses keywords in place of

punctuation and does not include any support for inheritance. The former introduces annoying syntactic differences between JJ and Java that make the transition to Java difficult, while the latter prevents OO programming concepts from being introduced before the transition to Java.

2.3 Language Levels

Simplified pedagogic languages based on Java have had limited success because they defer the teaching of Java and OO programming concepts. In these curricula, students simply learn procedural programming as preparation for learning Java, which is inconsistent with the progression of programming concepts taught in the Rice programming curriculum and other "objects-first" approaches to teaching introductory programming. After students learn a pedagogic language, they must still confront the task of learning Java and OO programming concepts.

A more ambitious and effective approach to lowering the learning curve for OO programming in Java is to define a series of progressively richer subsets of the Java language and develop an environment to support them. The concept of language levels dates back to at least the 1970's with the SP/k [17] language hierarchy. However, the design and implementation of a programming environment for a sequence of language levels for a real programming language requires a major investment, so few implementations of the concept have reached production code status.

2.3.1 SP/k

SP/k [17] is a sequence of language levels for PL/I developed by R.C. Holt *et al* at the University of Toronto in 1973. The goals of the SP/k project were to encourage structured problem solving by computers, to eliminate confusing and redundant language constructs, and to make the language easy to compile. SP/k consists of a hierarchy of eight subsets, each of which adds new programming language constructs to the preceding level. The array of language features covered by these subsets begins with arithmetic and variables, progresses to include control flow and loops, and ends with procedures, records, and files.

The SP/k processor was designed to function in a batch-processing world, where students prepared programs on punched cards. As a result, the processor tried to maximize the useful information in student programs by isolating errors encountered during compilation, temporarily repairing them on the fly, and completing the compilation and execution process. The compiler generated diagnostics reporting all the errors that were found, encouraging students to correct repaired errors rather than leave them in their programs.

2.3.2 DrScheme

DrScheme [12] is a pedagogic IDE for Scheme developed by Matthias Felleisen and his students at Rice University. DrScheme provided the initial inspiration for

DrJava and pioneered the form of interactive user interface used in DrJava. Like DrJava, DrScheme includes a source language editor, an integrated compiler and an Interactions Pane where users can evaluate arbitrary Scheme expressions using the functions that they have defined. However, the auxiliary tools supported by the two IDEs are quite different. Unit testing is a central feature of DrJava while it is provided by a third party as a plugin for DrScheme. DrJava has a comprehensive source level debugger while DrScheme has a stepper that allows users to walk through the execution process of some programs.*

DrScheme supports a hierarchy of language levels where each level in the hierarchy expands the scope of data abstractions and computations supported by the language. In contrast to earlier formulations of language levels like SP/k, the language hierarchy in DrScheme is based on semantic restrictions rather than syntactic ones. For example, the Beginning and Intermediate language levels in DrScheme ensure that data is immutable. The language constructs included in each level are dictated by the data model corresponding to that level. At the Beginner level, only algebraic data types are supported. At the Intermediate level, the data model is enriched to include functions as data. Finally, at the Advanced level, data mutation is introduced.

The language levels in DrScheme are driven by a comprehensive pedagogy of

*Stepping is restricted to programs at the Beginning and Intermediate language levels because the rewriting semantics on which stepping is based becomes very complex in the presence of data mutation.

program design that is largely language independent. This pedagogy is explicated in the book *How to Design Programs* [11], which uses Scheme as the vehicle for teaching program design. Fortunately, the same ideas can be applied to almost any type-safe language that supports functions as data. The design of our language levels facility for DrJava is based on this observation.

2.3.3 ProfessorJ

ProfessorJ [13] is a plug-in to DrScheme that supports a small subset of the Java language. ProfessorJ is implemented on top of the same MzScheme base as DrScheme instead of a conventional JVM. Therefore, there is no support for the standard Java libraries, severely limiting the subset of Java can be explored in the context of ProfessorJ—regardless of language level. ProfessorJ is still under development; the current release is rather primitive, but presumably will be improved as the implementation becomes more mature. In its current state, the ProfessorJ editor does not support either forward brace matching or auto-indenting. Except for java.lang.Object and java.lang.String, none of the built-in classes are supported (for example java.lang.Integer). ProfessorJ supports Beginner, Intermediate, and Advanced language levels of its own design with a "Full" language level in an alpha stage. We will discuss the differences between our language levels and ProfessorJ in Chapter 4.

Chapter 3
DrJava

3.1 Rationale for Use

We have chosen to implement our language levels facility as an extension of DrJava for several reasons. First and foremost, our development team has extensive experience developing DrJava, so we are intimately familiar with both the code base and design philosophy. Second, DrJava's focus on simplicity is in line with our goals. We want students who have never programmed before to quickly become acclimated to the tools used for instruction. DrJava has all of the features necessary for a novice:

- an editor with syntax highlighting, brace matching, and indenting;
- an integrated compiler;
- an integrated testing framework; and
- a workbench (the Interactions Pane) for performing interactive computation—bypassing the ugly Java command line interface—enabling beginners to experiment with their own code as well as with the Java libraries.

Finally, language levels fit naturally into DrJava's *textually* based interface. Although each language level is semantically motivated, it is precisely defined in syntactic terms because syntax is far more concrete and accessible than semantics. Language levels

simplify the task of mastering the syntax of Java because the simpler levels enormously reduce the available range of syntactic forms. It is conceivable that the same ideas could be adapted to a visually based environment like BlueJ, but the visual interface would have to modified so that the restrictions for each language level can be formulated in visual terms.

DrJava is stable, and its *test-driven* development process is supported by the use of open source software engineering tools including Ant [2] for build scripting, CVS [8] for a source code repository, and JUnit [3] for a unit test framework [28]. These tools facilitate further development of DrJava and contribute to its robustness.

3.2 Integration with DrJava

DrJava differs from most programming environments in that it supports the development of programs that merely consist of a collection of files. DrJava does not require the user to set up a *project* stipulating exactly which files belong to a program. DrJava infers entries for the program CLASSPATH (the list of directories potentially containing program files) based on the location of each open file and its package name.* As a result, simply opening a file in DrJava automatically makes any other files placed in the same package hierarchy available to the environment for purposes of compilation and execution.

*Java forces the directory structure above program files to mirror the structure of the package name to which the file belongs. See section 7.2.1 of the Java Language Specification [32]. Given this convention, it is easy for the compiler and the JVM to find external classes.

In integrating language levels with DrJava, it is important to preserve transparent access to the file system containing the program. In addition, files written in one language level should interact seamlessly with files written in other language levels or full Java.

To preserve transparent access to the file system, we store language level code in files just like regular Java source files but with different suffixes. When a language level source file is compiled, we generate a conventional source file with the standard .java suffix, including any required code augmentation corresponding to the language level file. In order to differentiate between files containing language level code and regular Java source files, we use the suffixes .dj0, .dj1, and .dj2 for Elementary, Intermediate, and Advanced files, respectively.

To set the current language level for the environment, we added a menu item that enables the user to specify the current language level, which can be any of our three levels or Full Java. Although each file's language level corresponds to its suffix, the choice of current language level in the DrJava environment forces new untitled files to conform to the specified language level (when compiled) and to be saved with the corresponding suffix. It also filters the open file dialog to only include files with this suffix. As a result, the Java source files generated for a language level file (which are given the standard .java suffix) are hidden from the user while using language levels within DrJava. On the other hand, these files are available for use with the Java

command line interface if the user so chooses.

Since the code augmentation process generates equivalent Java source files for language level files, programs can transparently intermix files written in different language levels. They seamlessly interoperate with other because they all correspond to conventional Java source files.

Chapter 4
Language Level Design

4.1 Overview

In our experience, Java's complex syntax interferes with students' learning of OO programming principles. To tame this complexity, we have broken down the language into a hierarchy of sublanguages. Many of our design ideas were expounded in early form in Brian Stoler's thesis, "A Framework for Building Pedagogic Programming Environments" [31] and an unpublished paper written by Cartwright, Nguyen, and Wong, "Can We Teach OO Design to Beginners?" [5].

We established four overall goals for our language levels facility:

1. Supporting a natural progression of language subsets for teaching OO programming in Java, following the pedagogy presented in *How to Design Programs* [11]. This progression should enable students to learn only the syntax relevant to the concepts that are currently being taught, making OO programming in Java accessible to beginners unfamiliar with basic language constructs.

2. Generating language-level specific error messages that refer only to the syntactic features of the given level. This policy should enable students to understand all of the diagnostics generated for language level code.

3. Enabling students to follow good programming practice–both in terms of design

and process–from the very beginning of the curriculum. By restricting students to a small portion of the language at a time, a curriculum can enforce much stricter stylistic and design standards.

4. Automatically generating the boilerplate code implied by the programming model for each level. This code augmentation process should make it much easier for beginners to define their own classes, and it should significantly reduce the amount of syntax that they have to learn. *All of the code generated for a given level is implied by the invariants underlying the programming model for that level.*

Practically speaking, these four goals dictate that we define the following specifications for each language level:

- the permitted language constructs;
- the automatically generated code; and
- the wording of error messages so that they are meaningful to the user.

The primary inspiration for our design of language levels is the progression of program design concepts in the Rice introductory programming curriculum, as codified in the textbook *How to Design Programs.* This progression consists of three major stages:

1. Elementary: functional programming using simple, inductively-defined (algebraic) data;

2. Intermediate: functional programming with functions as data; and

3. Advanced: imperative programming involving the mutation of data structures and variables.

When this curriculum is recast in object-oriented form, each of the three stages reduces to learning and mastering a new family of *design patterns.*

In the OO programming community, design patterns are generally regarded as important because they identify effective design practices, provide a common vocabulary to describe design, and help document class relationships [10]. From our perspective, design patterns play an essential role in OO programming pedagogy because they identify the coding patterns that correspond to specific programming models. In essence, these design patterns provide a vocabulary for defining program abstractions. Each of the language levels in our hierarchy corresponds to a richer vocabulary for defining program abstractions.

Since the Rice introductory programming curriculum is partitioned into three discrete stages, each corresponding to a different programming model, our hierarchy of language levels consists of three corresponding levels—Elementary, Intermediate, and Advanced—as defined in the following three sections.

4.2 Elementary Level

In essence, the Elementary level supports functional programming in Java. A functional program is a collection of equations defining a set of functions. The program is executed by simplifying an expression constructed from constants, primitive operations, and the functions defined in the program to an irreducible form, typically a data value. Functional programming is easy for beginning students to learn and master because the computations can be described completely in terms of a step-by-step source-level evaluation process that reduces an expression to an answer. Students learn to conceptualize computation as as an elegant series of algebraic simplifications rather than a long sequence of low-level operations that modify the contents of machine memory and registers [6].

Functional programs manipulate *algebraic* data types, which are sets of data values that have inductive definitions that can be formulated as tree grammars [33]. For example, a natural number can be zero or the successor of any other natural number. A function that accepts a natural number must account for the cases where the natural number is zero or when it is any number greater than zero. This correspondence between the definition of a data type and the shape of a function that processes the data type forms the basis of a powerful programming method called "data-directed design" [11]. Indeed, this observation provides the appropriate high-level structure for functions that can guide beginning programmers in designing programs rigorously

and analytically [6].

With the exception of arrays, all of the data types encountered in typical introductory programming assignments are algebraic: unbounded integers, booleans, lists, and trees. Students can immediately begin learning OOP principles because the natural OO representation for algebraic data uses the *composite pattern* [16]. Similarly, the natural format for defining operations on composite data is the *interpreter* pattern. Finite algebraic types are best represented using the *union* pattern, a simplification of *composite*. The *factory method* pattern can also be taught at this level.

The defining characteristic of functional programming is the immutability of program data. This restriction supports a simplified programming model that reduces reasoning about programs to familiar algebraic reasoning from mathematics where "equals can be freely substituted for equals." For example, in the absence of mutation, the expression fact(a) * max (b, c) can be evaluated by substituting the values of a, b, c into the expression and using a series of simplifying rewrites to produce the correct value. For example, assume that a = 3, b = 2, and c = 1. Then we can immediately reduce the expression to fact(3) * max(2, 1). Then fact(3) can be reduced (in a series of steps) to the value 6 and max (2, 1) can be reduced to 2. Then 6 * 2 is reduced to 12. This form of evaluation should remind students of the algebraic rules for simplifying expressions that they already learned in grammar and secondary school.

Mutation is potentially dangerous because if the value of an object is changed, all references to that object are also changed. Mutation breaks abstractions such as that of using objects as keys in hash tables and the sharing of tree nodes to save space and time.

Another benefit of enforcing immutability is that since fields cannot be mutated, they are implicitly final, implying that all fields must be initialized by the class constructor. Hence, the form of the constructor is completely determined, just as it is in functional languages. Our language level framework automatically generates this constructor for each of the user's classes.

We support Java 1.5 autoboxing and unboxing in all our language levels because these conventions narrow the gap between primitives and objects, eliminating the need for writing annoying manual boxing and unboxing code in nearly all contexts. Note that we provide this functionality only when DrJava is run using a Java 1.5 JVM, because prior versions of the JVM do not provide the requisite compiler or library support.

Since arrays are inherently imperative (involve data mutation), we prohibit them in both the Elementary and Intermediate levels. In addition, we also ban loops at the Elementary and Intermediate levels since they too are inherently imperative.

We further simplify the Elementary level by banning the use of package and import statements. The absence of these statements does not preclude the use of programs

with multiple files, but it forces all program classes to reside in the default package. This convention postpones the need to teach students how the files for a package-based programs must be stored in the file system. Since all program classes must reside in the default package, import statements are of little use.

In addition, the Elementary and Intermediate language levels do not include the null constant (a keyword). This policy prevents programs from generating null-pointer exceptions, eliminating the need to mention exceptions at the Elementary language level.*

For the sake of simplicity, all access specifiers (final, protected, private, synchronized, **volatile**) are prohibited. All fields, local variables, and parameters are implicitly private and all methods are implicitly public. These conventions can be taught to beginners without mentioning explicit access modifiers, introducing the concept of information hiding.

Finally, to eliminate confusion between class members and object members and to discourage procedural coding idioms, the Elementary level bans the use of the static keyword. This convention means that the *singleton pattern* cannot be taught until the Intermediate level.

To support the idea that classes define algebraic data types, the Elementary lan-

*This is true unless students access some of the more obscure parts of the java.lang library. Instructors should strongly discourage students from using any library functions that return null at the Elementary level.

guage level automatically generates the following methods for each user-defined class:

- Accessor methods for each field.
- A constructor that takes a value for each field as an argument.
- An overriding definition for the toString method.
- An overriding definition for the equals method.
- An overriding definition for the hashCode method.

Code augmentation plays a critical role in our language level hierarchy because the generated code cannot be expressed either as Elementary or Intermediate level code yet is generally necessary in order to form legal Java programs! The details of the augmentation process are described in Section 5.7.

4.3 Intermediate Level

The Intermediate level expands the functional programming model to include code (closures) as data. In a functional language, this generalization of the functional programming model simply involves introducing explicit λ-notation for functions. In Java, the same generalization involves adding support for anonymous inner classes and interfaces. Since anonymous classes have only a single instance (which is identified with the class definition), anonymous inner classes can be used as values in computations. The *command* and *strategy* design patterns codify this programming

technique. In both patterns (which are very similar), program execution depends on the run-time type of an abstract class or interface.

Since almost all of the interesting programming examples at this level involve algebraic data represented using the composite pattern, it is natural to introduce the *visitor* pattern as an enrichment of the strategy pattern (over a composite) that enables a programmer to decouple the definitions of operations on an algebraic type from the definition of the type.

Introducing package and import statements provides access to the Java API libraries. Since some of these library methods are declared to throw exceptions, it is necessary to introduce exceptions at this level. Hence, the Intermediate level includes the try-catch construct, the throw construct, and the throws qualifier in method declarations.

The Intermediate level also introduces explicit visibility specifiers (public, private, protected) for classes and methods as another step toward full Java. Since the libraries are exposed at this level and make extensive use of visibility modifiers, the introduction of these declaration qualifiers cannot be deferred any longer. Note that all fields are still implicitly final.

Students can begin to use casts at this level since the visitor pattern often requires casts. Many of the Java API library classes also require the use of casts. Adding full-support for generics may let us delay the introduction of casts in the future.

This level also supports the declaration of auxiliary constructors. Although code augmentation still generates the standard algebraic constructor with an argument for each field, students have the option of defining additional constructors that take fewer or different arguments and invoke the algebraic constructor by invoking this(...). For example, students could define an auxiliary constructor for a non-empty list class that takes a single argument and constructs a single element list containing this argument.

Finally, since we believe that the *singleton* pattern is fundamental and should not be deferred until the Advanced level, the Intermediate level also supports the static modifier for *fields.** In contrast, static methods are still banned because they embody procedural coding.

Code augmentation for the Intermediate level is identical to the Elementary level except that access modifiers are no longer generated automatically. The access modifiers in the generated Java code are identical to those appearing in the corresponding Intermediate level source files.

4.4 Advanced Level

The Advanced language level supports the commonly-used core of Java. The only features that are not supported are the keywords and classes related to concurrency (synchronized, volatile, `Thread`, `Runnable`), initializers, native methods, bitwise operators, and labeled statements. Fields and variables are mutable at this level and

*The singleton patterns uses a static final field to hold the only instance of the defined class.

arrays, static methods, and loops are supported. At this point, students should be sufficiently adept at OO design that they can use procedural constructs with discretion. The Rice curriculum provides very specific guidance on when procedural coding is appropriate.†

The Advanced level accommodates nearly all of the remaining common design patterns [16] including *state*, *decorator*, and simple versions of *model-view-controller* that only rely on the event thread for execution after setup. In our view, introductory courses do not need to venture beyond this level because general multi-threaded programming is deep, complex subject that is best deferred until a more advanced course devoted to the subject. The other restrictions imposed by the Advanced level, with the possible exception of bitwise operators, are rarely used in introductory courses. The Advanced level excludes the bitwise operators to prevent students from accidentally using them in place of the common && and || operators.

Note that the Advanced level supports named inner classes which are similar to anonymous inner classes except that they are explicitly named and may be instantiated more than once.

Code augmentation is done on a class-by-class basis at the Advanced level. If a class represents immutable data, it can be designated as algebraic which triggers the

†Roughly speaking, procedural coding is required when defining operations on a class that is closed and does not support a visitor interface. The built-in Java array types are prominent examples of such classes. In addition, procedural coding may be justified when the corresponding OO code requires excessive stack space or requires significantly more code; the use of loops is frequently justified on this basis.

same code augmentation that is performed at the Intermediate level.

Language Construct	L1	L2	L3	Full
Classes	X	X	X	X
Non-void methods	X	X	X	X
fields and local variables	X	X	X	X
`abstract` modifier	X	X	X	X
int, **double**, **char**, and **boolean** types	X	X	X	X
Simple arithmetic/comparison operators	X	X	X	X
if statement	X	X	X	X
Explicit constructors		X	X	X
package and **import** statements		X	X	X
static fields		X	X	X
Anonymous inner classes		X	X	X
Casts		X	X	X
Visibilty modifiers for methods		X	X	X
null value		X	X	X
Exceptions		X	X	X
Interfaces		X	X	X
Assignment to fields and local variables			X	X
Assignment operators (=, +=, ++, etc.)			X	X
Explicit use of **final** modifier			X	X
Nested classes and nested interfaces			X	X
while, **for**, and **do** loops			X	X
switch statement			X	X
void methods			X	X
Arrays			X	X
break and **continue** statements			X	X
All other primitive types				X
Static and instance initialization blocks				X
native methods				X
synchronized, **volatile**, **Thread** classes				X
Bitwise operators				X
Labeled statements				X
Conditional operator				X

Table 4.1 A table showing the availability of language constructs at the different language levels in our implementation.

Augmentation	L1	L2	L3
Methods, classes are automatically **public**	X		
Fields are automatically **private**	X		
Automatic constructor generation	X	X	
Fields, variables, and parameters are automatically **final**	X	X	
Automatic generation of **toString**, **equals**, and **hashCode**	X	X	
Automatic generation of accessors	X	X	
Concrete methods are automatically **final**	X	X	

Table 4.2 A table showing the automatic code augmentations that are made at each language level.

Elementary	Composite Interpreter Factory
Intermediate	Command Strategy Visitor Singleton
Advanced	State Decorator Model-View-Controller

Table 4.3 A table showing the OO design patterns that can be taught at each language level.

4.5 DrJava vs. ProfessorJ

There is a major difference in the design philosophy of language levels between DrJava and ProfessorJ. Although ProfessorJ is being developed under the guidance of the principal author of DrScheme and a co-author of the *How to Design Programs* textbook codifying the Rice introductory programming curriculum, it does not adhere very closely to the progression of programming concepts presented the book. ProfessorJ supports three language levels—called Beginner, Intermediate, and Advanced—plus a "Full" language level*.

In ProfessorJ, constructor definitions are required at all language levels, which adds considerable clutter to class definitions. At the Beginner level, constructors may contain field assignments. No code augmentation is performed. Mutation is introduced at the Intermediate language level, yet closures (anonymous inner classes) are not supported in any language level short of the Full Java level. No inner classes of any kind are allowed at any language levels except for "Full" Java. Similarly, the `static` keyword is deferred until the Advanced level preventing the use of the *singleton* pattern at the appropriate point in the curriculum. Since anonymous classes are not supported until Full Java, ProfessorJ prevents the critically important *visitor*, *command*, and *strategy* patterns from being taught in any generality prior to the "Full" Java level.

*Which of course is a misnomer because none of the Java libraries, including `java.lang` are supported.

Chapter 5
Language Level Implementation

5.1 Overview

In this chapter, we describe how we have implemented the language levels facility described in the previous chapter. For each language level, the implementation consists of a processor (translator) that performs the following four steps:

- Scan a source file written for the given language level to ensure that it conforms to the syntactic restrictions of that level.
- Augment the code with the automatically generated methods and modifiers specified for that level.
- Display meaningful errors messages if any program errors are encountered.
- Compile the file if there are no errors.

The processor for each language level initially reads a program (represented as ASCII text) from a source file and parses it into an *abstract syntax tree* (AST) representation, which is well-suited to syntax-checking, type-checking, and code augmentation. The AST data type is represented using the *composite* pattern. Each different form of tree node in an AST is represented by a distinct class extending the abstract AST class. Each node class represents a different language construct or syntactic element in a Java program. A node class contains fields which may be references to child

nodes in the tree. For example, there are distinct node classes that represent integer literals, method definitions, and parameter lists, respectively. A method definition node contains a reference to a parameter list which itself is represented by a node.*

It is straightforward to define methods that process AST's using either the *interpreter* pattern or the *visitor* pattern. In the interpreter pattern, a definition of the method must be included in every AST class. If a program defines very many operations on AST's using the interpreter pattern then the code for the AST classes becomes very cluttered and difficult to manage. The visitor pattern solves this problem by defining a visitor interface that includes a method for each different AST type and defining a visit (apply) method in each AST class. When the visit method is passed a visitor as an argument, it selects the appropriate method from the visitor (the method in the visitor interface for processing nodes of the type of this) and passes this to it. For more details on the visitor pattern, consult a book on OO design patterns such as [16].

In constructing the language levels implementation, we leveraged two program generation tools, ASTGen and JavaCC, that simplified the development process. These tools are described in the next two sections.

*Each node class has fields containing the coordinates (line and column number) for the node in the source file where it is located. These coordinates assist us in giving better error diagnostics because we can highlight the incorrect code snippet in the user's source code.

5.2 ASTGen

The definition of an AST data type for a production language like Java contains dozens of node subtypes implemented as concrete classes extending a general AST abstract class or interface. All of these node classes must implement the methods common to the AST data type including methods like toString() and equals(Object o) inherited from the universal Object type. As a result, significant changes in the design of the AST can force massive modifications to each node's class.

To address this issue, two of the original authors of DrJava, Brian Stoler and Eric Allen, developed a program generation tool called ASTGen. ASTGen reads an input file containing a tree grammar syntax defining the AST type hierarchy, generates the class definitions for the given types as well as visitor interfaces for traversing the AST, making sure the specified inheritance hierarchy is created. By automating the generation of the node class definitions and providing a single point of control (the input file) for defining the AST type hierarchy, it enables developers to modify the form of individual nodes or the entire AST type hierarchy simply by editing the tree grammar in the ASTGen input file. ASTGen is available under the GPL [14] license at **http://www.sourceforge.net/projects/astgen** [24].

5.3 JavaCC

Given a tool to generate the AST data type definitions, we also had to identify a tool for producing a parser to transform program source files into their AST repre-

sentations. The DrJava project had used the JavaCC parser generator in the past, so it was the natural choice for our parser. This parser generator takes a grammar and some supplementary information as input and creates a parser that processes a text file to determine if it conforms to the rules of the grammar. If so, it returns the AST. Otherwise, the parser returns an error message identifying the first input token that is not part of a legal program prefix. Using JavaCC simplifies the task of writing a parser and facilitates making changes to the grammar.

The major weakness of JavaCC is its poor error-detection and recovery system. Once the parser encounters a token that does not fit the grammar, it reports an error and abandons parsing. The generated error diagnostic is often difficult to understand, listing all possible language constructs that can appear at the point the error occurred. Since JavaCC only identifies the first error encountered in a source file, it forces users who have multiple syntactic errors to repeat the compile and debug cycle many times to correct all the mistakes. To alleviate this problem, we include "error productions" in our grammar to recognize syntactic errors, generate intelligible diagnostics, and continue parsing to process the entire source file.

Supporting a hierarchy of language levels significantly complicates the parsing process. We cannot simply use a parser for the full Java language to parse source files for each language level because such a parser would detect and explain syntax errors with respect to the entire Java language. A diagnostic may refer to constructs that

are foreign to a particular language level. For example, in a program with a missing closing brace for a class definition, a subsequent class definition will be interpreted as an inner class even though neither our Elementary or Intermediate levels include named inner classes. To produce appropriate error diagnostics, the parsing process must be tailored to each language level.

On the other hand, developing a completely separate parser for each language level would force the implementation to include three separate but very similar parsers, creating a level of code duplication inconsistent with good software engineering practice. Our solution to this conundrum is to define a common "coarse" grammar that embodies only the structure common to all of the language subsets. This grammar includes error diagnostics that explain errors in terms intelligible to users of any language level. These diagnostics do not mention any constructs not present in the Elementary level, and thus are suitable for any language level. The parser that we use is a modification of the Java parser developed by students in the Comp312 course at Rice in the 2003 Spring semester. It is available under the LGPL [15] license at **http://www.sourceforge.net/projects/javalangtools**.

An interesting issue is how to represent programs that have been parsed by our "coarse" parser. We cannot construct conventional Java AST's because our "coarse" grammar does not provide sufficient detail to build this form of representation. In essence, the coarse parser recognizes the lexical bracketing in a program and the Java

constructs that interact with bracketing. We call the representation that we build to represent Java programs at this level of detail JExpressions.

5.4 JExpressions

Brian Stoler explored the idea of JExpressions in his Master's thesis [31] in his discussion of possible future support for language levels in DrJava. JExpressions are similar to Lisp S-expressions in that they identify the lexical nesting structure of a program. JExpressions differ from S-expressions in the fact that they are crude AST's rather than lists. The major features of Java program syntax—including class definitions, field and method definitions, and if, while, for, do, and switch statements—are explicitly represented within a JExpression tree. But unparenthesized expressions are left as lists as tokens. Parenthesized expressions are aggregated into nested lists.

We decided to parse all language level source text into JExpressions rather than parse them into detailed Java ASTs using a full language level parser or separate level-language-specific parsers for the following reasons. Full Java parsing is unacceptable because the error diagnostics are not language-level specific. When a full parser encounters a syntax error in language-level code, some prefix of the erroneous code may be legal in full Java, producing an incomprehensible diagnostic when it finally encounters program text that makes no sense in full Java. We rejected writing a separate parser for each language-level, because it would introduce significant code duplication and make the code base difficult to maintain and modify. Our JExpres-

sion parser recognizes coarse language features (lexical nesting structure and related features) that are common to all of our language levels. Since the error messages returned by the JExpression (coarse) parser only pertain to features common to all language levels, they can be expressed in a form that is intelligible at any language level. An ancillary benefit of JExpressions over full Java ASTs is that JExpressions can be traversed by visitors with far fewer methods than visitors for Java ASTs. The full Java language is much more complex than the Elementary or Intermediate language levels. Hence, using a single full language parser would force the AST-walking syntax checker for each language level to cope with the syntactic complexity of the full language embodied in the visitor interface for the AST.* In the future, we anticipate that the JExpression framework may be useful in extending DrJava, because it could function as part of the code representation behind brace matching, indenting, and perhaps code-completion and the jump-to-definition feature.

The major disadvantage of using JExpressions is that the syntax-checking visitors still have to parse the token streams for expressions. These visitors can be tedious to code.

To simplify the task of manually parsing the token streams for expressions embedded in JExpressions, we have added some extra structure to JExpressions beyond what Brian Stoler described in his thesis. Our extension also accommodates the proper

*This complexity could be ameliorated by using the proxy pattern to define default behavior for all of the irrelevant node types.

parsing of Java generics because we anticipate that language levels will be extended to encompass generics once they are widely accepted by the Java user community. In the (coarse) JExpression parser, we match occurrences of < with > only when these signs appear in the context of generic types. In order to know when < is being used as the less-than operation as opposed to the end of a generic type parameter, we must parse statements and expressions where generic types can occur. For example, in the statement Vector v = new Vector<Integer>();, we cannot tell that the angle-brackets are being used around a type parameter without first knowing that this line of code is a statement, and more specifically, a variable declaration where v is being initialized to a new instance of a given type, namely the Vector type parameterized by Integer.

The list of Java constructs that can contain an instance of a generic type parameter includes class definitions, method definitions, parameter lists, field or variable definitions, allocation expressions, instanceof expressions, and cast expressions. Each of these constructs must be parsed in order to have the contextual knowledge necessary to determine the role of each occurrence of <.

Also, the grammar contains the notion of braced bodies. These are used for the bodies of class initializers, class bodies, and method bodies because they share many characteristics and, therefore, our visitors can share code. For instance, each braced body contains the list of variables defined in its scope and can be used when looking up the type of a given variable.

The grammar resulting from adding the rules necessary to parse these extra constructs is now much less lightweight and is only coarser than a full grammar for Java in that most strings of expressions are not parsed but rather are stored into an array of "expression pieces" in the AST. For example, `5 + foo.getFirst() + x` is not parsed at this stage because a generic type cannot appear here.

The resulting grammar is still rather coarse relative to a grammar for full Java, but it contains enough structure to reduce the manual parsing required and accommodate the notation for generic types. One consequence of being finer-grained than the original JExpression grammar is that the set of possible parsing errors is significantly larger. As a result, to support an informative user interface, we have to include many productions to cope with parsing program text containing errors. In essence, we have to handle most syntax errors internally within the grammar rather than relying on JavaCC's crude error reporting system.

5.5 Syntax Checker

Assuming there are no parsing errors, the parser generates a JExpression AST. Then the language processor for the given language level applies a visitor to the AST that determines if the the AST is well-formed (conforms to language level restrictions) and builds a symbol table for subsequent type-checking. In other words, the syntax-checking visitor for each language level performs the following two functions:

1. Check that all of the constructs appearing in the AST are allowed in this lan-

guage level, generating a diagnostic for each violation.

2. Populate a symbol table mapping class names to class attributes for use by the type-checker in a subsequent traversal.

The first function is easy to write because the visitor for the AST includes a method for every possible AST construct. For a given language level, the methods that correspond to prohibited constructs generate an error diagnostic each time they are called.

The second function requires that we create a symbol table that maps fully qualified class names to what we call ClassData objects. ClassData objects contain all the information about a class that is necessary for subsequent type-checking: name, modifiers, superclass, interfaces, fields, methods, outer class, and inner classes/interfaces. There are two sources for generating ClassData objects. One is an already existing Java class file, and the other is the AST produced by our parser.

To support the construction of ClassData objects from class files, we use the open-source Apache tool BCEL to read class files. To read a class file using BCEL, we add the class file's root directory to the classpath and invoke the BCEL class reader which returns a *JavaClass* object for the file. This object contains all of the information in a ClassData, so we simply have to translate this JavaClass to the equivalent ClassData. This process proceeds recursively, linking the ClassData objects for classes to those of classes they reference.

If no class file exists for a user-defined class or if the source file is newer than its class file, we use our parser to create an AST for the corresponding source file. Our visitor visits each class definition in the AST, creating a new ClassData for it and any class that it references. This process of creating the ClassData from a class is called *resolution.*

When resolving classes, other classes that are referenced are merely assigned a stub ClassData object that we call a *continuation.* A continuation only contains a fully qualified name. The reason we cannot immediately resolve these member types is that their resolution may refer back to one of the classes that we are currently trying to resolve. For example, assume class A extends class B and class B has a field of type A. If we start out resolving A, we will normally create a continuation called B. However, if we try to resolve B before completing A, we will then try to resolve A again, creating an infinite loop. The continuations that we create are stored in a list. After visiting the classes in the current file, we resolve each class in this list to make sure that the every entry in the symbol table is fully resolved.

The syntax-checking visitor examines the superclass and interfaces of each class defined in the source file and then moves on to each field, method, and inner class of the class, filling in the information of the ClassData. Once again, if we encounter a type that is not already in the symbol table, we must create a new continuation for it. It is important to stress that fully qualified names must be used for all entries in

the symbol table to distinguish classes with the same names but different packages.

We use the following series of questions, derived from Section 6.5 of the Java Language Specification [32], to find the fully qualified name given an unqualified class name and return the corresponding ClassData object.

1. Is this class name is a primitive type? If so, return the singleton ClassData object that corresponds to the primitive type.

2. Is the class an array type? If so, resolve the element type.

3. Is the class to be fully resolved? If not, return a continuation.

4. Is the class specifically imported? If so, look for the class file, read it using BCEL and return the resulting ClassData object. If the class file does not exist or is out of date, visit the source file.

5. If the class name is qualified, does the user have access to it? If not throw an error. If so, return the ClassData object.

6. Is the class in the same file as the class we are currently parsing? If so, skip ahead in the file to the class and resolve it, later finishing up the current class.

7. Is the class in the same package as the class we are currently parsing? If so, read the class file or visit the source file.

8. Is the class in one of the imported packages? Read the class file or visit the source file.

9. Is the class in the java.lang package? Read the class file.

10. The class cannot be found; throw an error.

Since this algorithm is the same for all language levels, we have placed the method that implements it in a superclass common to the syntax-checking visitors for all language levels. We call this class LanguageLevelVisitor. It contains this method along with some fields that are common to all language level visitors including the symbol table, the list of syntax errors, and the list of continuations.

After applying the appropriate syntax-checking visitor to the AST, we display any errors that have been generated. If there are none, we continue on to the type-checking pass.

5.6 Type Checker

To determine the type of a given variable, the type-checker must keep track of its context, that is, what class or method body it is currently visiting. If the variable is within a method body, we must first check if it is a local variable. For each method the user writes, we create a MethodData that stores information about its local variables and inner classes. These MethodDatas are linked to their enclosing ClassDatas, which in turn are linked to their super classes and interfaces. The existence of this network

means that to find the type of an occurrence of a variable name, the type-checker simply has to recursively search through all the scopes that are visible from the point where the variable occurs. If the variable cannot be found, the type-checker generates an error because it is not visible from the current scope.

The general idea of type-checking in Java is to examine any place where the user:

- Assigns one type to another
- Uses an operator
- Invokes a method that takes arguments
- Casts one type to another
- Performs a field access
- Defines a concrete method. If declared to return `void`, the method can only contain void return statements. If declared to return a value, all branches of code flow must return the correct type or throw an exception.
- Throws an exception. The user must either declare that the method throws the exception, or must catch it at the invocation site.

The type-checking visitor traverses the AST until it reaches a node containing one of these cases. It looks up the types of the variables involved by searching for the declaration of each variable in the current scope and then by looking up the type

in the symbol table. These types are compared for compatibility with each other given the operator or method that is using the types. This process usually involves finding if a type is a sub-type of another one since when a certain type is required, generally speaking, any sub-type will work instead. Our design of the symbol table facilitates this lookup process by linking each ClassData object with its super class and interface. Checking for compatibility involves performing simple walks up the inheritance chains.

An interesting case is the cast expression: Integer x = (Integer) o;. In this example either o must be a subclass of Integer or Integer must be a subclass of o. In the former case, this cast does nothing, but still type-checks correctly. This case requires two walks up inheritance chains to type-check.

Another task the type-checker must perform is ensuring that method bodies return the correct type. To accomplish this we introduce the notion that when visiting statements, the type-checking visitor returns a type. The returned type depends upon the kind of statement. For a return statement, it is the type of the value being returned (or void if none is returned). For a block statement (i.e. a series of statements enclosed by braces), it is the type of the first statement to return a type. For an if-then-else statement, it is the common value returned by both branches. If only one (or no) branch returns a type, then since we cannot know with certainty which path will be taken, we must return null, indicating that no conclusion can be drawn. A

try-catch statement works that same way. In all other cases, the return types are null. This protocol guarantees that a method will always return a type and that the type is correct. It also accounts for the checked exceptions that can be thrown by ensuring that they are either contained within a try-catch or a method that is declared to throw the exception. Another benefit of this protocol is that it is simple to detect that a statement is unreachable since this is the case for all statements in a block following the first statement that returns a type.

5.6.1 Type Checking Error Diagnostics

It is easy for the type checker to generate intelligible error diagnostics because the messages are independent of the language level and simply explain a type clash. In fact, there is only one type-checker for all language levels since the type-checking process is the same for every level. If there are no errors, then processing moves on to the final stage.

5.7 Code Augmentation

Code augmentation is the process of adding in access modifiers and helper methods to the user's source code. Augmentation is necessary at the Elementary level and Intermediate language levels because both language levels elide "boilerplate" code. Automatic generation of constructors and field accessors is essential at the Elementary and Intermediate levels since there is no other way to initialize object fields. In

addition, we generate a definition of toString to make the string representations of user-defined classes more intelligible, a definition of equals to support algebraic comparison (rather than object identity) and hashCode to enforce the Java policy that two objects that are equal also have the same hash code.

The code augmentation is done as a mapping from the original AST to an augmented AST. We create a copy of the original AST, making changes to the nodes that require the addition of visibility modifiers and adding methods to the nodes for classes. To generate the augmented source file, we simply unparse the AST (walking the AST outputting the corresponding source text).

The source code for language levels can be found at **http://www.sourceforge.net/projects/javalanglevels**. It is available under the DrJava Open Source licence [20].

Chapter 6
Programming Methodology

6.1 eXtreme Programming

Writing and maintaining reliable software is difficult; bugs can creep into programs of any size very easily. Bugs are particularly pernicious in tools designed for beginners for two reasons. First, beginners have difficulty distinguishing aberrant behavior by the tool from mistakes in their input to the tool. Thus, bugs in DrJava may lead beginners to believe that they are making mistakes in their programs. Second, beginners are much more likely to use unorthodox command sequences and strange coding conventions, stressing a tool in ways that the developers failed to anticipate and exposing hidden bugs.

Our approach to ensuring program reliability is to follow the core principles of Extreme Programming (XP) during the coding process: *simplicity*, *incremental, test-driven development*, (implying *continuous integration*, *small releases*, and *unit testing*), *pair programming*, and the presence of an *on-site customer* [22].

6.1.1 Simplicity

When solving a problem, we generally favor the most straightforward approach. This strategy allows us to produce working code faster which can then be easily tested and refined. Also, given the likelihood of future modifications to the original

problem specifications, trying to perfect a segment of code may end up being a waste of resources. In the language levels project, we focused on developing one language level at a time instead of trying to build a general framework with all of the specific functionality that would eventually be required.

6.1.2 Incremental, Test-driven Development

When writing a program, it is beneficial to produce a working prototype as soon as possible, enabling users to provide immediate feedback. Small improvements are continually added and new versions made available to the customers. Introducing each new feature or bug fix as it is finished is called *continuous integration.* This process allows users to focus their evaluation on one new feature at a time and helps isolate the source of new bugs. These *small releases* help users track the progress and direction of the developers.

One way users can provide specifications for a program is by providing sample inputs and their corresponding outputs. When creating language levels, we used *unit tests* as a way to guide our coding and increase its robustness. Unit tests check that a given method or larger code unit has the expected behavior on some sample inputs. Using JUnit [3], we have written a combination of top-down tests, where we input a source file and test that the correct errors are generated and the file is properly augmented, and bottom-up tests, where we test individual methods for correctness. Because of the unusual and unexpected syntax that beginners may write for language

levels, simply testing the code by using our examples is not sufficient. Bottom-up testing is valuable because it involves selection of test cases that together cover the entire code base, as opposed to just testing the code's functional specification. Using both approaches is important because it provides it confirms that each code unit performs as intended by the developer and that the entire program performs as expected by the customer [26]. This testing process took us a great deal of time, but in the end, we found it was worth the effort. Covering the code base with tests makes it much easier to modify the program, because bugs created by additions and revisions will be immediately detected because some unit test will fail. Unit tests also provide a way for those unfamiliar with the code to learn how it works by observing what the code is designed to do given certain inputs [21].

One of the precepts of unit testing is that the code quality is proportional to the ease of testing. We found this principle to be helpful because it forced us to re-write and improve some of our code in order to make it more testable. The method described earlier that finds the qualified class name given an unqualified one and the current context was extremely difficult to test, so we broke it down into several methods and tested those individually.

6.1.3 Pair Programming

An undergraduate and I wrote most of the code for language levels together. We routinely switch off between "driving" and "navigating". The driver types while the

navigator sits next to the driver correcting typographical errors, suggesting better ways to write a unit of code and identifying possible problems. Pair programming is effective because of the extra creativity, clerical precision, and code auditing that it brings to the coding process. In addition, pair programming helps foster group ownership of the code base. If one developer leaves a group, there is at least one other person with a detailed knowledge of each block of code that the departing person helped write. Pair programming can similarly be used to help new developers learn the code base. This form of knowledge transfer is invaluable since code must be continually maintained and updated.

6.1.4 On-Site Customer Input

While program correctness can be measured objectively, it is not easy to measure how effectively the design of language levels conforms to the pedagogy presented in *How to Design Programs* [11].

Since the introductory curriculum at Rice University follows the pedagogy set out in *How to Design Programs*, professors and students at Rice are the natural initial customers for the language levels facility. Early versions of language levels have been tested by students and faculty in our JavaPLT research group at Rice and by students in Comp312, an undergraduate course that teaches production programming. This summer, we unveiled language levels at a workshop called *TeachJava* for high school AP teachers from around the country; the workshop showed teachers how to use

language levels to support an AP curriculum similar to what we teach in the first year Computer Science course at Rice. In the Spring of 2005, Comp201—the first semester of our new all Java introductory course sequence—will make extensive use of language levels because it is aimed at teaching students who are not Computer Science majors how to program in Java. We anticipate that language levels will make learning OO programming more entertaining and easier to grasp by students with little or no programming background.

Chapter 7
Future Work

The language levels facility now meets all of our original design goals, but we will continue to refine and extend the program to meet the needs of our customers. From our early experience and customer feedback, we need to continue working on providing clear, unintimidating error diagnostics reported using a friendly interface. We also want to explore the possibility of supporting a configurable language levels facility capable of supporting other pedagogies for teaching OOP.

7.1 Parser Error-Recovery

We plan to improve our coarse Java grammar to enable our coarse parser (generated by JavaCC) to recover gracefully from most syntactic errors and complete parsing the source file, returning useful information about all of the errors in the coarse structure of the input. We also anticipate leveraging this modified parser in DrJava's editor to support code completion (popping up a list of all the type-correct method and field names for a receiver expression) as well as jumping to the definitions of field and method names that appear in program text.

7.2 Java 1.5 Extensions

We plan to support generics and other Java 1.5 extensions in appropriate language level dialects. We have made a provision for adding support for generics by modifying

our parser to parse generics and storing the type parameters in the symbol table. The remaining challenge in supporting this extension is extending our type-checker to handle generic types.

As for other 1.5 features, we have already implemented autoboxing/unboxing in all language levels. DrJava only enables this support if the selected compiler is 1.5 compatible. We also plan to support foreach, static import, variable arguments, and enumerations, but we have not yet fully determined in which level each feature should be introduced.

7.3 Configurable Language Levels

Our ultimate goal is to develop a configuration facility for language levels, which would empower instructors to design language levels based on their own pedagogy rather than ours. One of the main issues that we will have to confront is whether we can support code augmentation in the presence of mutation because it breaks invariants on which our current code augmentation depends. At a minimum, we should be able to support configuration options that do not affect the status of mutation.

Appendix A
Example of Code Augmentation

A.1 Elementary level code

```
abstract class IntList extends Object {
  abstract int sum();
}

class Empty extends IntList {
  int sum() {
    return 0;
  }
}

class Cons extends IntList {
  int first;
  IntList rest;

  int sum() {
    return first + rest.sum();
  }
}
```

A.2 Augmented version

```
abstract class IntList extends Object {
  public abstract int sum();

  public IntList() {
    super();
  }
}

class Empty extends IntList {
  public int sum() {
    return 0;
  }
```

```
  public Empty() {
    super();
  }

  public String toString() {
    return "Empty(" + ")";
  }

  public boolean equals(Object o) {
    if ((o == null) || getClass() != o.getClass()) return false;
    return true;
  }

  public int hashCode() {
    return getClass().hashCode();
  }
}

class Cons extends IntList {
  private final int first;
  private final IntList rest;

  public int sum() {
    return first + rest.sum();
  }

  public Cons(int first, IntList rest) {
    super();
    this.first = first;
    this.rest = rest;
  }

  public int first() {
    return first;
  }

  public IntList rest() {
    return rest;
  }
```

```
  public String toString() {
    return "Cons(" + first + ", " + rest + ")";
  }

  public boolean equals(Object o) {
    if ((o= null) || getClass() != o.getClass()) return false;
    Cons cast = (Cons) o;
    return first == cast.first && rest.equals(cast.rest);
  }

  public int hashCode() {
    return first ^ rest.hashCode();
  }
}
```

References

1. E. Allen, R. Cartwright, B. Stoler. *DrJava: A lightweight pedagogic environment for Java. SIGCSE Technical Symposium on Computer Science Education.* Sept. 2001. **http://www.drjava.org**

2. The Apache Software Foundation. "The Apache Ant Project." **http://ant.apache.org**

3. K. Beck, E. Gamma. "JUnit, Testing Resources for Extreme Programming." **http://www.junit.org**

4. Bloch, Joshua. **Effective Java** *Programming Language Guide.* Addiston-Wesley, Boston, MA, 2001.

5. R. Cartwright, D. Nguyen, S. Wong. *Can We Teach OO Design to Beginners?.* Proposal, June 2003.

6. R. Cartwright, M. Felleisen, A. Papakonstantinou. *Functional Programming: An Active Approach to Learning Algebra.* NSF CRLT Proposal, 1996.

7. Chatley, Robert. *Kenya.* Master's thesis. **http://chatley.com/kenya/thesis/other/**

8. Concurrent Versions System. **http://www.cvshome.org**

9. R. L. Constable and R. W. Conway. "PLICS – A Disciplined Subset of PL/I." Technical Report 76-293. Department of Computer Science, Cornell University.

10. Eliëns, Anton. *Principles of Object-Oriented Software Development.* Addison-Wesley, Harlow, England: 2000. p 60.

11. M. Felleisen, R.B. Findler, M. Flatt, S. Krishnamurthi. **How to Design Programs**, *An Introduction to Programming and Computing.* MIT Press, 2001.

12. R. Findler, C. Flanagan, M. Flatt, S. Krishnamurthi, M. Felleisen. *DrScheme: A pedagogic programming environment for Scheme.* Integernational Symposium on Programming Languages: Implementations, Logics, and Programs, 1997, 369-388.

13. K. Fisler, M. Flatt. *ProfessorJ: A Gradual Intro to Java through Language Levels. SIGCSE 2003*, October 2003.

14. Free Software Foundation. The General Public License.
www.gnu.org/licenses/gpl.html

15. Free Software Foundation. The Lesser General Public License.
http://www.gnu.org/licenses/lgpl.html

16. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, Mass. 1995.

17. R.C. Holt, D.B. Wortman, D.T. Barnard, J.R. Cordy. "SP/k: a system for teaching computer programming." *Communications of the ACM*, Vol. 20, Issue 5. May 1977. pp 301-309.

18. Holt Software. "Ready To Program with Java Technology."
http://www.holtsoft.com/ready/

19. Hong, Jason. "The use of Java as an introductory programming language."
ACM Crossroads, Summer 1998.
http://www.acm.org/crossroads/xrds4-4/introjava.html

20. JavaPLT. DrJava Open Source License. **http://drjava.org/license.shtml**

21. R. Jeffries. "Essential XP: Documentation."
http://www.xprogramming.com/xpmag/expDocumentationInXP.htm

22. R. Jeffries, A. Anderson, C. Hendrickson. *Extreme Programming Installed.* Addison-Wesley, 2001.

23. M. Kölling, A. Patterson, B. Quig, J. Rosenberg. "BlueJ, The Interactions java Environment." **http://www.bluej.org**

24. Open Source Developer Network. "SourceForge." **http://sourceforge.net**

25. Pattis, Richard. *Karel the Robot.* New York: Wiley, 1981.

26. D.E. Perry, G.E.Kaiser. *Adequate Testing and Object-Oriented Programming. J. Object-Oriented Programming*, Jan./Feb. 1990. p. 13.

27. PublicStaticVoidMain.com. "JJ Home Page."
http://www.publicstaticvoidmain.com

28. Reis, Charles. *A Pedagogic Programming Environment for Java that Scales to Production Programming.* Master's thesis, April 2003.
http://drjava.org/papers/creis-thesis.pdf

29. E. Roberts. "An Overview of MiniJava." In *Technical Symposium on Computer Science Education*, 2001, pp. 1-5.

30. D.H. Steinberg, *The effect of Unit Tests on Entry Points, Coupling, and Cohesion in an Introductory Java Programming Course.* Proceedings of XP Universe Conference'01, 2001.

31. Stoler, Brian. *A Framework for Building Pedagogic Java Programming Environments.* Master's thesis, April 2002. **http://drjava.org/papers/bstoler-thesis.pdf**

32. The Java Language Specification **http://java.sun.com/docs/books/jls**

33. Tommasi, Marc. "Definitions." **http://l3ux02.univ-lille3.fr/ tommasi/TATAHTML/node27.html**

34. Holt Software. "What's New on the Holt Software Web Site." **http://www.holtsoft.com/whatsnew.html**