# TOWARDS CUSTOMIZABLE PEDAGOGIC PROGRAMMING LANGUAGES

by

Kathryn E. Gray

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

August 2006

Copyright © Kathryn E. Gray 2006

All Rights Reserved

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

# SUPERVISORY COMMITTEE APPROVAL

of a dissertation submitted by

Kathryn E. Gray

This dissertation has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

______________ ______________________________
Chair: Matthew Flatt

______________ ______________________________
Wilson Hsieh

______________ ______________________________
Gary Lindstrom

______________ ______________________________
Joe Zachary

______________ ______________________________
Gilad Bracha

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

# FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the dissertation of Kathryn E. Gray in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date

Matthew Flatt
Chair, Supervisory Committee

Approved for the Major Department

Martin Berzins
Chair/Dean

Approved for the Graduate Council

David S. Chapman
Dean of The Graduate School

# ABSTRACT

In introductory education, pedagogic tools and languages can help focus students on the fundamental programming concepts. Subsets of a professional language support the presentation of the language's syntax in step with the presentation of the underlying concepts, relieving students and instructors from superfluous details. This dissertation presents the design and implementation of pedagogic tools integrated with an introductory computer science course.

The tools described in this dissertation include languages, compilers, and libraries tailored to improve introductory education for students and instructors. Specifically,

- three pedagogic subsets of the Java programming language. My method of designing these subsets, through a combination of assessing the supported curriculum and observation, may serve as a guide for creating other subsets. Discussion includes experiences using these subsets in classrooms, and how observations guided the design and modifications of the subsets.
- an extension to Java supporting fine-grained interoperability between languages, which facilitates the reuse of existing libraries accessible within pedagogically appropriate subsets of a language. More generally, I demonstrate how to support general interoperability between two languages. Discussion includes the interoperability mechanism and a library implementation using the language extension.
- an examination of the compiler implementation supporting the subsets and interoperability extension. The compiler maps Java constructs into similar Scheme constructs for ease of use of Java programs within Scheme and vice versa. Discussion includes parsing multiple languages with coherent error messages, representing Java compilation units, and supporting dynamic checking of Scheme values in Java.

To my parents

# CONTENTS


**ABSTRACT** . . . . . **iv**

**LIST OF FIGURES** . . . . . **ix**

**LIST OF TABLES** . . . . . **x**

**ACKNOWLEDGEMENTS** . . . . . **xi**

**CHAPTERS**

**1. LANGUAGE SUPPORT FOR PEDAGOGY** . . . . . **1**

1.1 Support from Language and Environment . . . . . 2
1.1.1 Language Support . . . . . 2
1.1.2 Environment Support . . . . . 4
1.2 Pedagogic Libraries . . . . . 4
1.3 Customizing Languages . . . . . 5
1.4 Towards Customization . . . . . 5
1.4.1 Pedagogic Subsets . . . . . 6
1.4.2 Programming in Multiple Languages . . . . . 7

**2. CREATING PEDAGOGIC LANGUAGES** . . . . . **9**

2.1 A Curriculum and Its Requirements . . . . . 9
2.2 Observations . . . . . 11
2.2.1 Observations Using Non-ProfessorJ Compilers . . . . . 11
2.2.2 Observations Using ProfessorJ Compiler . . . . . 16
2.3 General Lessons . . . . . 18
2.4 ProfessorJ Languages . . . . . 20
2.4.1 Beginner . . . . . 20
2.4.1.1 Ongoing observation. . . . . . 23
2.4.2 Intermediate . . . . . 24
2.4.2.1 Ongoing observation. . . . . . 25
2.4.3 Advanced . . . . . 26
2.5 Connecting to the Environment . . . . . 29
2.6 Supporting Error Messages . . . . . 31
2.7 Related Work . . . . . 33
2.7.1 Language Levels . . . . . 33
2.7.2 Pedagogic Languages . . . . . 35
2.7.3 Pedagogic Tools Without Language Levels . . . . . 35

**3. ADDING LIBRARY SUPPORT** . . . . . **37**
3.1 Language Mixings . . . . . 38
3.1.1 A Dynamically-Checked Pedagogic Language . . . . . 38
3.1.2 Parametrically Polymorphic Library . . . . . 39
3.1.3 A Dynamically-Typed Library . . . . . 41
3.2 Extending the Language with `dynamic` . . . . . 41
3.2.1 Syntax and Extensions . . . . . 41
3.3 Writing Libraries with `dynamic` . . . . . 43
3.3.1 Dynamically-Checked Student Language . . . . . 43
3.3.2 Polymorphic Library . . . . . 44
3.4 A Teaching Library . . . . . 44
3.4.1 The Library . . . . . 45
3.4.2 Implementation Options . . . . . 45
3.5 The Implementations . . . . . 47
3.5.1 Image . . . . . 47
3.5.1.1 Native Method Implementation. . . . . . 47
3.5.1.2 Java + `dynamic` Implementation. . . . . . 49
3.5.1.3 Discussion. . . . . . 49
3.5.2 View . . . . . 50
3.5.2.1 Native Method Implementation. . . . . . 50
3.5.2.2 Java + `dynamic` Implementation. . . . . . 50
3.5.2.3 Discussion. . . . . . 53
3.5.3 GameWorld . . . . . 53
3.5.3.1 Native Method Implementation. . . . . . 53
3.5.3.2 Java + `dynamic` Implementation. . . . . . 55
3.5.3.3 Discussion. . . . . . 55

**4. EXPLORING DYNAMIC** . . . . . **57**
4.1 Type Checking `dynamic` . . . . . 57
4.2 Protecting Type-Safety . . . . . 60
4.2.1 Contracts . . . . . 60
4.2.2 Contracts and Java + `dynamic` . . . . . 61
4.2.3 Checking Contracts . . . . . 63
4.3 Adding Support for `dynamic` . . . . . 67
4.3.1 Mirrors and Reflection . . . . . 67
4.3.2 Implementing `dynamic` with Mirrors . . . . . 68
4.3.3 Static Types from Dynamic Values . . . . . 70
4.4 Related Work . . . . . 71
4.4.1 Mixing Dynamic and Static Types . . . . . 71
4.4.2 Language Interoperability . . . . . 72

**5. BUILDING THE SYSTEM** . . . . . **74**
5.1 Compiling for Multiple Languages . . . . . 75
5.1.1 Parsing . . . . . 75
5.1.2 Type Checking Multiple Languages . . . . . 76
5.1.3 Error Messages . . . . . 76
5.2 Type Checking `dynamic` . . . . . 77
5.3 Compiling to Scheme . . . . . 78

5.3.1 Compilation Units . . . 78
5.3.2 Classes . . . 81
5.3.3 Fields and Methods . . . 82
5.3.4 Nested Classes . . . 84
5.3.5 Statements and Expressions . . . 85
5.3.6 Native Methods . . . 88
5.4 Compiling for Interoperability . . . 88
5.4.1 Objects and Methods . . . 88
5.4.2 Fields . . . 89
5.4.3 Inserting Dynamic Checks . . . 89
5.4.4 Supporting Equality and Casting . . . 92
5.4.5 Inheritance . . . 92
5.4.6 Naming Conventions . . . 93
5.5 Java–Scheme Data Mapping . . . 94
5.5.1 Strings . . . 94
5.5.2 Arrays . . . 94
5.5.3 Exceptions . . . 95
5.6 Related Work . . . 95

**6. IN THE FIELD . . . 98**

6.1 Anecdotal Assessment . . . 99
6.2 ProfessorJ in the Classroom – Self-Evaluation . . . 99
6.2.1 Teacher's Perspective . . . 99
6.2.2 Students' Perspective . . . 101
6.2.2.1 Survey Responses. . . . 101
6.2.2.2 Personal Comments. . . . 102
6.2.3 Conclusions . . . 103
6.3 ProfessorJ in the Classroom – External Evaluation . . . 103
6.3.1 Teachers' Perspective . . . 103
6.3.2 Students' Perspective . . . 106
6.3.3 Conclusions . . . 110
6.4 Developing Libraries for ProfessorJ . . . 110
6.5 Onwards . . . 111
6.5.1 Specification Language . . . 112
6.5.2 Separating Error Messages . . . 113
6.5.3 Parsing . . . 114
6.5.4 Type Checking and Compiling . . . 116
6.5.4.1 Error Messages. . . . 116
6.5.4.2 Modifying the Back-end. . . . 117

**APPENDICES**

**A. DRAWING LIBRARY . . . 118**

**B. USER SURVEYS . . . 136**

**REFERENCES . . . 140**

# LIST OF FIGURES

1.1 Java programs calling Scheme . . . . . 8
2.1 ProfessorJ Beginner program within DrScheme . . . . . 30
3.1 Combining dynamic pedagogic language with Java . . . . . 39
3.2 Combining polymorphic library with nonpolymorphic language . . . . . 40
3.3 Drawing library: Student interface . . . . . 46
3.4 Image : Native methods . . . . . 48
3.5 Image: Java + `dynamic` . . . . . 49
3.6 View: Native methods . . . . . 51
3.7 View: Java + `dynamic` . . . . . 52
3.8 GameWorld : Native methods . . . . . 54
3.9 GameWorld : Java + `dynamic` . . . . . 56
4.1 Checking flat values in Scheme . . . . . 61
4.2 Checking functions in Scheme . . . . . 61
4.3 Enforcing constraints with contracts . . . . . 62
5.1 Java program with dependency cycle . . . . . 80
5.2 Compiling Java classes . . . . . 82
5.3 Compiling overloaded methods . . . . . 84
5.4 Java class and wrappers for `dynamic` . . . . . 90
5.5 Implementing fields in wrappers . . . . . 91

# LIST OF TABLES

2.1 ProfessorJ Beginner Declaration Constructs . . . . . . . . . . . . . . . . . . . . . . . . 21
2.2 ProfessorJ Beginner Statements and Expressions . . . . . . . . . . . . . . . . . . . . . 22
2.3 ProfessorJ Intermediate Declaration Constructs . . . . . . . . . . . . . . . . . . . . . . 24
2.4 Intermediate Java Statements and Expressions . . . . . . . . . . . . . . . . . . . . . . . 25
2.5 ProfessorJ Advanced Declaration Constructs . . . . . . . . . . . . . . . . . . . . . . . . 26
2.6 ProfessorJ Advanced Statements and Expressions . . . . . . . . . . . . . . . . . . . . 27
4.1 Positions to Insert Checks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
4.2 Positions to Insert Guards . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
6.1 Teachers Using ProfessorJ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 104
6.2 Students' Assessment of Overall Experience . . . . . . . . . . . . . . . . . . . . . . . . 107
6.3 Beneficial Features of ProfessorJ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 108
6.4 Problematic Aspects and Features of ProfessorJ . . . . . . . . . . . . . . . . . . . . . . 109

## ACKNOWLEDGEMENTS

Thanks to my advisor, Matthew Flatt, for all his guidance and advice, for teaching me how to give a talk, and for his efforts to secure funding for my research.

I appreciate the comments and time given to me by all of my committee members.

To the students and teachers using ProfessorJ, I owe you my deepest gratitude. And you also have my deepest apologies for any bugs encountered. I especially thank all of those who participated in the surveys.

My co-authors of the *How to Design Classes* and the authors of *How to Design Programs* deserve strong recognition. Without your efforts this work would have encountered many more obstacles. Additionally, to my co-developers of PLT Scheme, my software could not stand without yours.

Thanks to my family, friends, and of course my husband, Scott.

And a special thanks to Dan Friedman and Matthias Felleisen — without the *Little Schemer*, I would not have taken that first step towards becoming a computer scientist.

# CHAPTER 1

# LANGUAGE SUPPORT FOR PEDAGOGY

Introductory programming frustrates many students. The syntax overwhelms, the error messages confuse, and too much time passes before the program does something real. Instructors confront these problems by structuring the course to follow the needs of the programming language and spending extra class time to explain language minutia and error messages.

Many difficulties stem from the use of professional programming languages and compilers. Professional languages, including C, C++, Java, etc. and their compilers were developed for use by fully trained programmers. Such tools allow students to (accidentally) write programs they cannot be expected to understand and report mistakes using terminology the student cannot be expected to know.

However, switching to a pedagogic language and compiler does not necessarily ease the confusion and frustration. Pedagogic languages, such as Jeroo, Alice, and Logo, were designed for use by novice programmers with little to no prior training. These languages suffer from one of two problems: either they contain enough constructs to fill a course, which leads to the same problem as professional languages; or they do not contain enough and must be abandoned, requiring the students to move to a professional language too soon. In addition, instructors and students do not have the same wealth of libraries available as those using professional languages, because fewer people write libraries in pedagogic languages.

Using one pedagogic language is not the answer; using a professional language is not the answer; using several subsets of professional languages, where each subset adds more features of the professional language, provides a solution that mixes the best of pedagogic and professional languages.

Another approach mixes professional and pedagogic tools at a different level by providing a pedagogic graphical interface on top of a professional compiler and language. The interface can provide a set of tools to bypass language constructs, additional ex-

planations of common error messages, or exploratory tools to guide students in creating and understanding their program. Despite these additions, the underlying reliance on a professional compiler allows students to encounter many of the same problems as with using all professional tools. Syntactically correct, but unintentional and misunderstood programs can be created. Also students can make mistakes that fall outside of the common error messages and encounter bizarre messages they cannot understand.

Pedagogic development environments do provide useful tools and a more controlled experience for students. Pedagogic subsets best support students when coupled with a pedagogic environment. This combination provides student tailored interaction and language constructs, with error messages written to their understanding. As the student matures and needs new language constructs, instead of learning an entire new language, their overall knowledge of the language grows. Further, existing professional libraries can be made available throughout an introductory course.

## 1.1 Support from Language and Environment

Pedagogic subsets provide a programming experience tailored to introductory students while allowing access to professional libraries and features when desired. Also, using the subsets adds complexity through the course without changing to a different programming language, requiring learning a new syntactic structure. Supporting the subsets within a pedagogic environment provides the support of a development environment without over burdening students. Features of the environment can also impact which language constructs are unnecessary.

### 1.1.1 Language Support

A pedagogic language subset contains only enough syntactic and semantic rules to support the current concepts being covered in class. Very few constructs not yet presented to the student appear in the language. This restriction not only prevents students from using constructs they do not understand, but also allows the compiler to present error messages that could not exist for a professional language, phrasing these messages in terms appropriate to the student.

With a Java subset without mutation, the compiler can inform students when a field value was not initialized within the constructor. A compiler for the full language cannot provide this error message, as fields can be initialized in many places and in many ways.

A student who encounters strange behavior based on the default value of a field may have difficulty tracing the error back to their failure to properly initialize all fields of the class.

In all introductory classes, students learn a subset of the language just large enough to begin writing programs. The professional compiler supports the entire language, and cannot inform students when they stray outside of the subset, whereas a compiler for the subset presented informs students when they stray, to avoid confusion and poor programming style.

In another example, an introductory student might write the following program:

```
int convert(int in, int by) {
... }
...
convert(4);
```

Most Java compilers respond with an error message reporting that the method `convert` cannot be found (sometimes including the type of the given argument in the message). Students who have not yet learned about overloading are baffled by the compiler's claim. A compiler for a language without overloading can report a clear message to the student.

A single pedagogic subset is insufficient for an introductory course. Like a pedagogic language, this subset cannot scale to support all of the concepts to be covered in the course. And like a professional language, this subset can be too large to allow tailored error messages. Multiple subsets provide the small languages necessary for specific error messages (and to prevent accidental use of advanced features) and the complexity for a course to scale up to the full language.

Such subsets, known as ***language levels***, exist for PL/I (SP/k) [1] and Scheme (DrScheme) [2]. These subsets demonstrate the viability of presenting a curriculum with a language-processor enforced restriction of a professional language. The SP/k system, constrained by the expenses of processor availability, did not attempt to address the confusion caused by error messages, instead opting to report as many errors as possible. The DrScheme system provides student-targeted error messages, but does not face the complications of type errors and inheritance inherent in Java. Neither prior work presents explanations for designing language levels (although Holt. et al present general criteria for pedagogic languages) nor addresses the impact of language levels in a classroom.

### 1.1.2 Environment Support

Using a professional language subset allows the compiler to produce targeted error messages. Using a pedagogic environment further reduces complexity and confusion for students, with a simplified interface and interactions designed for student exploration. Additionally, interactions supported through the development environment can (temporarily) replace constructs in the language.

For example, in the development environments DrScheme [2], BlueJ [3], and DrJava [4], students run their programs interactively. There is no need to dictate before compilation a series of statements and expressions to evaluate. Not only does this feature allow students to explore functions, methods, objects, etc., but it also eliminates the need for `public static void main( String[] args)`. With this need eliminated, the student language can safely eliminate static methods, void methods, and arrays without affecting the students ability to write and run usable programs.

## 1.2 Pedagogic Libraries

Basing pedagogic languages on a professional language, instead of developing a new language, allows teachers to leverage existing libraries in developing programming assignments. These libraries allow assignments to reflect more real-world situations and programming tasks. However, as a professional compiler is not compatible with pedagogic endeavors, professional libraries might not be compatible with classroom understanding. To circumvent problems with a professional library, an instructor-developed library – ***teaching library*** – provides a student-appropriate interface to the full library.

In the presence of a subset language, this teaching library must present an interface consistent with the pedagogic language. In fact, the teaching library should present very similar interfaces to all of the subsets. When the static semantics of the professional library and the pedagogic language do not match, the teaching library must bridge this gap with runtime checks and conversions.

Proper placement of such checks, especially for use with several different languages, is error prone. Thus supporting multiple pedagogic languages, while aiding in education, raises additional problems for instructors in creating teaching libraries.

In essence, the problem is supporting interoperability between two languages. Traditionally interoperability solutions access on language from the other language through an API. Methods provided by the API convert values, but the programmer must explicitly call them. Doing these conversions correctly is difficult and error prone. When interop-

erability is supported with a language extension, where the compiler inserts checks and conversions where needed, teaching libraries that bridge two (or more) languages become easier to create.

With a language extension, the programer denotes where the switch between different languages occurs (either through a specific name or type designation). The compiler uses the known type and language information to perform the necessary checks and conversions for the value at this position. The implementor cannot forget or sidestep a necessary check. Additionally, if an error occurs in the teaching language implementation, with compiler added checks, the library implementor will receive an error report that allows them to quickly address the problem.

## 1.3 Customizing Languages

A subset of a professional language must support only some language features; the choice of which features to support arises from the choice of curriculum to follow in the course. Thus, even when using such pedagogic tools, instructors must either follow the tool designers' curriculum, continue to contort their curriculum to meet various expectations of the tool, or create their own tool.

Tool creation takes time and expertise beyond the resources of most instructors. To wed a curriculum to a tool or language, the curriculum developer evaluates which concepts need to be supported and which cause students unacceptable levels of confusion. The developer then selects the language constructs to present to the student, and organizes the constructs into language sets. A compiler needs to be developed to support these sets with student-targeted error messages. Additionally, the compiler must interact with a development environment, and desired libraries must be incorporated into the curriculum, including creation of teaching libraries.

A system that accepts a declarative language specification to generate a student-oriented compiler will greatly reduce the effort required in creating or modifying specialized languages, allowing more instructors to undertake this task.

## 1.4 Towards Customization

In order to build a declarative compiler generation system, several preliminary steps are needed. These steps form the basis for this work, while a declarative specification remains for the future.

- Before creating a general mechanism to create pedagogic subsets, such subsets must be developed and studied. Otherwise, the resulting general specification mechanism might not support the restrictions (and extensions) desired in these languages. Additionally, study of these languages in classrooms will reveal the usefulness of these subsets and the nature of changes made over time. If subsets are not beneficial, the general mechanism would not be either.
- Additionally, instructors and students (indirectly) may need to interact with both the development environment and libraries written in the full language. Therefore, the compiler must support safe and simple interoperability between the development environments language, the full language, and the pedagogic language(s). While in many instances, the environments language is identical to the full language, this uniformity does not occur in all environments. Providing a general fine-grained interoperability solution provides support for library creation in either instance, as well as allowing the static semantics of pedagogic languages to vary from the full language if desired.

Both steps lead to the thesis that:

> Restricting a language through a pedagogic environment provides improved feedback to students, and fine-grained interoperability through language extensions supplies needed flexibility and support in implementing subsets and teaching libraries.

### 1.4.1 Pedagogic Subsets

As mentioned above, initial pedagogic subsets must be created prior to developing a general framework. Three subsets are presented in Chapter 2: Beginner, Intermediate, and Advanced. These subsets present Java to programmers with one semester prior programming experience[1].

Each subset adds features to the previous language, while presenting a language that is always statically and semantically similar to Java. Classroom usage, to be discussed further in Chapter 6, demonstrated flaws in the language design and curriculum choices. The initial languages were then redesigned to support changes to the subject matter treated. These steps help illustrate the benefits of a declarative specification while providing the first steps in implementing such a system.

---

[1]Although the subsets have been used in first semester courses.

### 1.4.2 Programming in Multiple Languages

In our environment, a teaching library may need to interoperate with a dynamically-typed library in addition to interoperating with full Java libraries. Without improvements to interoperability, teachers must employ standard techniques to access functionality and ensure proper treatment of the values.

Consider a dynamic library, written in Scheme, that provides functions to open a drawing canvas, draw lines in the canvas, and draw various shapes. In order for these features to be usable within a statically-typed Java-like language, a teaching library will need to bridge the gap between the Java-like language and Scheme.

The teaching library must either use an API (internally using reflection) to call into the dynamic library or implement the functionality through a native interface, producing code similar to the example seen in Figure 1.1(a). Objects entering the dynamically-typed library that originated in the student's program must be wrapped with an object that checks accesses on all methods to ensure proper treatment of the types. Failure to do this step will allow confusing runtime type violation errors to occur.

By supporting interoperability as a language extension, the bridge between the teaching language and the professional library becomes easier and less error prone, as can be seen in Figure 1.1(b). The compiler inserts conversions and checks into the program based on the type information of the program; thus checks cannot be overlooked and cannot be incorrectly applied.

Examples of using this language extension to provide teaching libraries between languages with different static-properties are presented in Chapter 3. Additionally, an extended example of a teaching library connecting Scheme to the existing language levels also appears.

Explanation of the type-system and insertion of coercions necessary to support this language extension is presented in Chapter 4.

Chapter 5 presents the implementation of the compiler, including discussion of the student-oriented error messages as well as implementation of the language extension. Chapter 6 presents the use of the pedagogic languages in various courses as well as discussion of extending the existing system to support customizable languages.

```
class View {

  scheme.interp.SchemeInterface scheme =
         scheme.interp.SchemeInterface.init(‘‘draw.lib’’);

  void display(int xSize, int ySize) {
    scheme.call(‘‘open-canvas’’, scheme.toObject(xSize),
                scheme.toObject(ySize));
  }

  boolean draw( Command c ) {
    Object result = scheme.call(‘‘draw-object’’, this.translateProtect(c));
    if (scheme.toBool(scheme.call("succeeded", result)) )
      return true;
    else
      throw new RuntimeException(scheme.call("explanation",result));
  }
}
```

(a) Interoperability with an API

```
import scheme.draw;

class View {

  void display(int xSize, int ySize) {
    draw.openCanvas(xSize,ySize);
  }

  boolean draw( Command c ) {
    dynamic result = draw.drawObject(c);
    if ( draw.succeeded( result ) )
      return true;
    else
      throw new RuntimeException( draw.explanation(result) );
  }
}
```

(b) Interoperability with dynamic

**Figure 1.1**. Java programs calling Scheme with different interoperability technologies

# CHAPTER 2

# CREATING PEDAGOGIC LANGUAGES

Designing subsets of a programming language takes into account a curriculum and the experience level of intended users. Effective pedagogic language levels require a close partnership with a curriculum. Otherwise, the language either adds inappropriate restrictions or it allows too many programs. Therefore, the first step in developing pedagogic languages is understanding the needs of the curriculum.

Further steps involve observing student programmers, defining the subsets to support the curriculum and common student errors, and then evaluating and revising both the subsets and the curriculum based on new observations. This chapter presents the development of the language levels for ProfessorJ [5] and its connections to the *How to Design Classes*(HtDC) curriculum.

## 2.1 A Curriculum and Its Requirements

The HtDC curriculum covers object-oriented programming in second semester courses using the ProfessorJ language levels. Both the curriculum and the language levels continue the curriculum of the first course, using *How to Design Programs* (HtDP)[6] and the corresponding Scheme language levels.

The first course covers program design in a largely functional style, using subsets of Scheme, with a data-centric approach. A "data-centric" approach means that the course first teaches students to understand the data representation for a problem, and then to allow the shape of the data to drive the rest of the design. A canonical example from early in the semester is the "list of numbers" datatype, which might be used to represent a list of prices in a toy store:

A *list-of-numbers* is either

- *empty*
- (`cons` *number list-of-numbers*)

The *list-of-numbers* data definition drives the implementation of an *inventory-value* function that consumes an instance of the datatype. In particular, the function's implementation should match the shape of the data: two cases, handling a compound data value in the second case and a self-reference in the second element of the second case:

```
(define (inventory-value l)
  (cond
   ((empty? l) ...)
   ((cons? l) ... (first l)
             ... (inventory-value (rest l)))))
```

This data-oriented approach in the first course transitions naturally to a similar "object-oriented" approach in the second course. A data definition with two clauses corresponds to an interface with two implementing classes. The two **cond** lines in *inventory-value* turn into separate method implementations in the subclasses, with a method invocation in the second one.

```
interface Inventory { int Value(); }

class Empty implements Inventory {
  Empty() { }
  int Value() { return 0; }
}

class Addition implements Inventory {
  int val;
  Inventory rest;
  Addition(int v, Inventory r) {
    val = v;
    rest = r;
  }
  int Value() { return val + rest.Value(); }
}
```

Initial data definitions in the second course are simpler than this example, fitting into one class with no interfaces. From this starting point, the course progresses by presenting more complicated forms of data including lists, trees, and acyclic graphs.

The first methods students write are functional methods that call no other methods and return testable values. With more complicated data definitions, students progress to writing methods that delegate work to other methods and that recur, still functionally.

As program complexity grows, the course introduces abstract classes and methods to avoid repeated code within a class hierarchy and local variables within a method. Once inheritance has been established with abstract classes, students learn about full inheritance and overriding the functionality of the super class's methods.

Next, the course moves to discuss mutation within an object-oriented setting, motivated by the creation of cyclic-graphs. Methods are no longer functional and can be used solely for their (potentially multiple) side-effects.

After mutation, overloaded constructors are introduced, followed quickly by access controls that allow constructors to be hidden. These lead to covering overloaded methods and the creation of libraries from students' programs. Students begin to use some standard Java libraries instead of teaching libraries. This leads to the discussion of arrays, loops, and iterators.

Depending on the duration of the course, the content finally covers parametric polymorphism, exception handling, i/o, and further abstractions using inner classes.

Several possible language divisions could cover this curriculum. However, regardless of where the exact divisions are drawn a few restrictions are key for this curriculum:

- The first language must support classes and forbid mutation.
- The language that introduces methods should require functional methods.
- The language that introduces overloaded constructors must also support access modifiers.

The ProfessorJ compiler covers this curriculum with three pedagogic languages, finishing with the full language. The first language presents the curriculum until abstract classes are introduced; the second presents the curriculum until overloaded constructors; the third presents material until any of polymorphism, exception handling, or inner classes are presented. More languages could also be used to narrow down the divisions in the curriculum; however, as the number of languages increases with a hard-coded implementation, the amount of maintenance and development time also increases significantly.

## 2.2 Observations

In addition to molding language levels to support a set of concepts, it is also important to remove troublesome, unnecessary constructs from early levels and to provide clear error messages for common confusing situations. Identifying such constructs comes through observation of students, with the correct background knowledge, programming with both existing tools (covered in Section 2.2.1) and the language subset compiler (Section 2.2.2).

### 2.2.1 Observations Using Non-ProfessorJ Compilers

For the development of the ProfessorJ languages, I observed students at the University of Utah taking CpSc 2020 using the BlueJ[7] development environment and a standard im-

plementation of javac during regular lab sessions, TA office hours, and in questions mailed to the 2020 course staff. At the time, CpSc 2020, the second course of the introductory sequence, presented Java to students familiar with Scheme and functional programming following the HtDP curriculum. Common problems, confusions, and behaviors were noted as areas the ProfessorJ language levels should address.

Through the rest of this section, specific error situations and the corresponding influence on the development of language levels are discussed.

***Observation*** 2.1 (**public, private, protected**). I observed that errors from typos in access modifiers (especially `public` at the start of a file) fostered considerable confusion among students, as well as confusion arising from the misuse of these modifiers.

Misspelling the first `public` results in an error message that includes the option to remove the modifier entirely, which many students chose to take as they did not notice their typo. This led to different problems later on when trying to run the class or access it externally. Some students learned (through practice) to randomly add or remove access modifiers in order to appease the compiler without understanding the meanings of their actions. Similar problems arose with the placement of `private` and `protected` on methods and fields.

Careful explanations can prevent problems with access modifiers, but the curriculum does not cover the distinctions until late in the course. Therefore, based on these observations, the initial language levels restrict access modifiers and treat all members as `public`.

***Observation*** 2.2 (**variable not found**). I observed that error messages reporting unbound identifiers when a field is incorrectly accessed as a method caused considerable confusion, as did the reverse situation. The students could see that the identifier was indeed bound, and did not notice that their usage was at fault. Once in this situation, the students could not deduce how to resolve the problem.

Both the HtDC curriculum and the 2020 curriculum explicitly presented the differences between fields and methods, but in these situations, students could not recognize the difference in their usage. This inability to visually distinguish the syntax suggested that a more detailed error message is required.

The ProfessorJ languages do not permit a field and method in the same class to have the same name. When a program accesses a field with parentheses, the compiler reports

that the field is incorrectly being accessed as a method and vice versa, to specifically draw the students' attention to the point of their mistake.

***Observation*** 2.3 (**if without else**). I saw that when students wrote programs similar to this example

```
if (condition())
  var = valTrue();
  var = valFalse();
```

they rarely noticed the lack of the `else` keyword between the two statements. Clearly, in the resulting program, `var` will always have the return from `valFalse()` without regard for the result of `condition()`.

I observed students attempting to resolve the error in their program by modifying the condition to always be true, testing the methods that created the values for `var`, printing out the result of `condition()`, and finally seeking help (occasionally proclaiming their computer to be broken, since all of the pieces worked individually). No student presenting this problem to me had even considered the possibility of a syntax error, and some did not believe that was the error even after correcting their program. These students believed that an incorrect program that correctly compiled could not be corrected by solely adding a keyword.

This potential error should be addressed when students learn that `if` statements do not require `else`. Before students specifically learn about this feature and the potential pitfalls, these observations indicate that the compiler should require the `else` keyword for all `if` statements. Since the HtDC curriculum does not cover this feature until late, the earlier language levels should restrict the statement.

***Observation*** 2.4 (**missing }**). Despite BlueJ tool support for creating classes that creates a new file for each class, I observed some students creating all of their classes in one file. Often these students omitted required parentheses and braces, typically a } to close a method. With multiple classes per file, the following circumstance can arise

```
class Empty extends List {
...

class Cons extends List {
}
```

The resulting error message, of a missing } to match the first { led many to place the missing } at the end of their file, inadvertently creating an inner class.

Since inner classes are legal Java programs, no error occurs until the student adds accesses to the `Cons` class outside of the `Empty` class. At this point, the error message indicates that the class `Cons` is unknown, resulting in similar confusion to that in Observation 2.2.

The problem occurs because introductory students are also novice brace balancers and tend to blindly follow the suggestions of the compiler. The situation can be avoided by either restricting the language to allow only one class per file, removing inner classes, or requiring students to use a class browser. Since the HtDC curriculum presents inner classes in the latter portion of the class, the ProfessorJ languages do not support inner classes and therefore avoid this confusion.

***Observation*** 2.5 (**static infection**). For the course I observed, students were required to include print outs of their examples and tests in the body of `public static void main(String[] args)`. Using static methods in order to perform tests and create executable content is not unusual. The students did not yet fully understand the difference between a static and instance method.

I saw many students calling nonstatic methods within the body of `main` without first providing an object. The compiler reported to them that a nonstatic method cannot be called from a static context. Often, students switched the method to be static to resolve the conflict, and then proceeded to change any fields or methods accessed by this method to static — a processes that continued until only nonused methods and the constructor did not contain a `static` modifier. Students were then befuddled by the incorrect behavior of their program that was intended to have per-object instead of per-class state.

This problem arises because the students do not understand the magic incantation of `public static void main...` and may forget the object of their method call. While both the BlueJ and DrJava [4] pedagogic environments eliminate the need for `main` to explore program behavior, many teachers still require the method for easier grading.

The HtDC curriculum does not cover static methods or per-class behavior until presenting access modifiers, therefore the first language levels do not need to support `static`. Like the BlueJ and DrJava environments, the DrScheme environment provides support for program exploration without using `main`. However, either the environment or the language will need to provide support for repeatable testing.

***Observation*** 2.6 (**more variables not found**). In addition to difficulties with unbound variable references when fields and methods are incorrectly accessed, students were also

confused with error messages arising from incorrect arguments in method calls. The error reports detail the method name and the arguments passed in, so that the programmer can compare these to the available overloaded methods.

When the student does not yet know about overloading, this is incredibly odd. Additionally, even when students have learned about overloading, they encountered difficulty in determining which portion of their method call was in error. The language level can confront this problem by withholding overloading until it will be presented in class, and then restricting the possibilities in order to provide clear error reports.

***Observation*** 2.7 (**inheritance and overloading**). Another difficulty I observed with overloaded methods occurred when students inadvertently created an overloaded method through inheritance. By accidentally omitting a parameter or modifying a parameter type, students create an overloaded method in the subclass instead of an overriding one.

In these situations, the students could not understand why the functionality of their new method did not occur when calling the method on instances of the class. They did not notice that the methods' signatures were not identical, especially since the two implementations were rarely simultaneously visible.

In languages without overloading, this problem does not arise as the methods of the super class constrain the available names and signature of the subclass. A clear error message is especially important in this circumstance, as the inherited method may not be in view. The ProfessorJ error message indicates the specific type signatures of the two methods and explains that they must match to override the method. In Java version 5, the error can also be avoided through use of the `@Override` annotation to force the compiler to check for this error, provided annotations fit within the pedagogic language.

***Observation*** 2.8 (**uninitialized variables**). Repeatedly, I observed students write programs with uninitialized fields where the student thought the field to be initialized in the constructor. The following code fragment demonstrates the style of program leading to this confusion:

```
class Car {
  String make;
  Car( String make ) {
    make = make;
  }
}
```

With this code, students expected their field to contain a usable value and could not understand the resulting error message, which indicated a null value for the `make` variable.

This problem can be avoided by forbidding assignment to local parameters, requiring all fields be explicitly initialized, or forbidding parameter names from hiding field names. The first restriction may restrict too many programs, in a language permitting mutation. The ProfessorJ languages avoid this problem by requiring that fields are explicitly initialized.

***Observation*** 2.9 (**reflection and professional libraries**). A different form of problem arose when I observed students discovering the many libraries available to Java programmers, especially libraries that partially implemented homework assignments and the reflection libraries. Many students used these libraries inappropriately for their assignments, to the detriment of their grade and understanding.

The reflection library allowed students, especially those confused by dynamic dispatch, to thwart the object-oriented nature of the programming language by writing methods that dispatched via `if` and reflective calls.

These problems can be addressed by lowering the students' grades for using the libraries; however, controlling the language can additionally allow the compiler to control the program's imports and prevent much consternation, arguments, and loss of learning opportunities for the students.

### 2.2.2 Observations Using ProfessorJ Compiler

Development of the ProfessorJ compiler did not end observations of the students. I observed students in the last three weeks of CpSc 2010 (University of Utah's introductory course) programming with the compiler, high school teachers (with little to no knowledge of Java) using the compiler during a one-week crash course, and I observed by-proxy (comments from course staff) students in other courses interacting with the tool. These observations led to refinements in the language level restrictions.

***Observation*** 2.10 (**condition always true**). In the language with assignment, the language has two very syntactically similar operators `=` and `==`. I observed students accidentally creating an assignment expression instead of comparing two values. This accident caused the greatest confusion in code similar to

```
if (val = false)
  trueMethod();
else
  falseMethod():
```

Clearly, the programmer intended to write `val == false`. In these circumstances, students did not notice they had mutated the variable they were testing, they only noticed that the method always selected one path of the `if` regardless of the value of `val`.

To resolve this confusion, the teaching languages do not allow assignment expressions even when assignment statements are permitted. An alternate possibility would restrict assignment from the conditional position of an `if` statement, allowing students to use the convenience of assignment expressions in other positions. This restriction, however, presents an inconsistent and confusing language to the students, which is undesirable.

Some teachers requested that using `==` to compare booleans be disallowed, in part to prevent this programming pattern. However, discussions with students showed that a reasonable proportion of the students following the programing convention find that declaring a variable equal to `true` assists them in reading and understanding the behavior of their program. As their comfort grows, they fall out of using this pattern, and so it should not be prohibited by the language.

***Observation*** 2.11 (**inheritance woes**)*.* In the original languages, fields of the subclass were allowed to hide the parent classes' fields. When inheritance was introduced, I observed students inadvertently hiding the parent classes' field names. While looking at the child class, the student did not recall that the parent class contained the same field.

As hiding had not yet been presented, the students could not understand why sometimes the value of their field contained the correct value and sometimes an incorrect value.

Resolving this confusion required either modifying the language or the curriculum, which was the undesired choice. Therefore, a subclass in the language level introducing inheritance cannot contain fields with the same names as the super class.

***Observation*** 2.12 (**curriculum changing observations**)*.* Other observations, such as confusions regarding where to initialize abstract classes' fields in the first language level, problems with super constructor calls, and super method calls, were initially addressed by adjusting the language levels. However, these problems were a symptom of a greater

confusion regarding the differences between extending an abstract class, implementing an interface, and extending a concrete class. Modifying the languages could not fully resolve these problems. Instead, the curriculum was restructured to present interfaces first, which made the language corrections based on these observations obsolete.

Since the language change, few problems in the list in the first sentence of this observation have recurred. However, more observations of students using the new language designs are still needed.

## 2.3 General Lessons

My observations of the difficulties students encountered while programming with Java led directly to the creation of the language levels presented in Section 2.4. In general, these observations also suggest guidelines that should be considered when adapting any professional language for pedagogical purposes. For practical concerns, it may be necessary for a pedagogical language to violate one of these guidelines. The language designer should be aware of the potential difficulties in doing this and construct error messages accordingly.

When creating the pedagogic languages, two general issues should be considered: that the intended progression of the languages does not add and then remove features; and that there are no program portions that will be explained to students using a "magic" explanation. In the first instance, adding a feature in one language level that will not appear in the next causes confusion and annoyance. Students become accustomed to using the syntax, and they are irritated by the removal and unlikely to accept the reasons considering their prior use of the construct. For the latter case, the students should not be exposed to features they are not expected to understand yet. A classic example is using `public static void main(String[] args)` without explaining the aspects of the interface.

From Observations 2.6 and 2.7, I conclude that overloading confuses students both on variance from expected behavior and naming concerns. Overloaded operators cause similar problems. For instance, in Java `+` and `==` perform different operations based on the types of the arguments. Context sensitive changes in behavior cause unnecessary difficulty. As a general guideline, overloading of any variety should not appear in a language for novices.

Several observations, including 2.2, 2.6, 2.7 and 2.11, demonstrate how name conflicts

cause confusion. When multiple entities within a program have the same name, students lose track of the individual meanings of the names and become frustrated. In general, a novice language should not permit multiple entities (including fields and methods, parameters and global variables, or classes and fields) to have the same name at the same time. However, when there is a curricular requirement for allowing name overlap, the general principle suggests that there should be a clear visible distinction between the names. For example, if a language allows field and method parameter overlap to discuss lexical scope, the fields must be accessed with `this` to distinguish them from parameters.

Observations 2.5 and 2.6 both indicate a general problem learning multiple forms of methods simultaneously. Distinguishing between static methods, instance methods, and separating methods by argument number and type — before fully understanding any one of these constructs — leads to confusion and misuse. As a general guideline, only one variation of a construct should be presented to students at a time, including only one of functions vs. methods, procedures vs. functions, or data-types vs. classes.

One of the primary causes of the problems noted in Observation 2.10 stems from the similarity between assignment `=` and comparison `==`. Especially in a language for students, syntactic forms should not be so similar. Other syntactic forms in the language are also too similar, including the syntax for calling a constructor and for instantiating an array. In general, a pedagogic language should not contain syntax that allows one construct to be easily mistaken for another. In a language where all syntax is similar (i.e., the syntax of Lisp or Scheme), the general guideline should preclude strong similarities between the names of operators.

Observation 2.3 indicates that students have difficulty with the optional `else` branch of an `if` statement. This syntactic form can also cause difficulties in writing a parser specification, requiring productions to resolve the ambiguities of the form. In general, grammars that increase the difficulty of writing a parser may be a bad choice for student-oriented programming languages, including such features as disambiguating operator precedence through position, similar structures for fields and methods, and the if without else problem.

Observation 2.8 demonstrates the confusion caused by allowing a field to be set with the default value. In Java, this situation is slightly better than in some other languages, whose specifications allow implementations to have unpredictable behavior when a variable is not explicitly initialized. In general, this situation should not occur in

introductory languages: a pedagogic language should not permit uninitialized values.

Following these criteria can lead to a pedagogic language that supports students while avoiding unnecessary confusion. However, these lessons form a guide and not unbreakable rules. Curricular needs and the base syntax of the professional language may require a language that contains syntax against these guidelines. In such cases, the guidelines suggest potential pitfalls that may require special attention in crafting error messages. For example, the ProfessorJ language levels do not avoid the similar syntax of the equality and comparison, because the courses using these languages stress the importance of using the Java language, but the error message support addresses the potential confusion.

## 2.4 ProfessorJ Languages

ProfessorJ presents three pedagogic language levels that support two-thirds of the second semester curriculum. The number of language levels is a compromise among three competing stresses: the difficulty of implementing and maintaining a language level (a stress that will be greatly reduced with the creation of a customizable system); presenting the material with small enough granularity to avoid confusing students; not overwhelming the students with too many language levels, which can make students feel insulted. As with any software project where human computer interaction is an integral portion, the psyche of the users must also be taken into account.

The three levels presented in this section are not the original levels that ProfessorJ supported, nor are they likely to be the final levels. As classroom experience regarding the levels and the curriculum grows, the content of the levels continues to shift to better support both the curriculum and the students' confusions. The paper "ProfessorJ: A Gradual Intro to Java through Language Levels" [5] presents the original language levels supported by ProfessorJ.

### 2.4.1 Beginner

The ProfessorJ Beginner language covers the curriculum from the introduction of Java until the introduction of inheritance. Students use the language to write simple classes with no methods or `implements` clauses through classes with recursive methods implementing an interface.

The declaration constructs presented in Table 2.1 provide enough structure to write small programs and experience the general flavor of Java.

**Table 2.1**. ProfessorJ Beginner Declaration Constructs

| Construct | Restrictions |
|---|---|
| imports | Some imports not allowed |
| classes | Implicitly public<br>Cannot be final<br>Cannot have static members<br>Cannot be extended |
| interfaces | Implicitly public<br>Cannot have static members |
| fields | Implicitly public<br>Implicitly final |
| constructors | Required<br>Implicitly public<br>Cannot be overloaded<br>May only contain assignments |
| methods | Implicitly public<br>Cannot be overloaded<br>Cannot return void |
| Exclusions:<br>package, inheritance, class and instance initialization blocks,<br>inner classes and interfaces, all modifiers, arrays | |

Students use classes to implement the same data structures they have already implemented in Scheme. These Scheme structures are groupings of data that map easily to objects with immutable fields, where the field values are given as constructor arguments. To facilitate this mapping, fields must be set in the constructor and cannot be further mutated. The curriculum also presents constant values, so fields can also be set immediately (and then cannot be modified). To avoid initialization order concerns, fields cannot be initialized using other fields.

Due to Observation 2.5, ProfessorJ Beginner excludes static members, and due to Observation 2.1 all other modifers have been removed as well.

In Java, methods and fields without any modifiers can be accessed by any other class in the same package. Since all classes are within the same file, they are also within the same package. Therefore, all members can be accessed by all other classes in the same program. However, when students override a `toString` method or implement an interface, where all members are implicitly public, the student's methods must also be public. Internally, the compiler treats all members as though the student declared them with a `public` modifier.

Although, with this technique, the students technically are writing incorrect programs, omitting the modifier until the Advanced language does not appear to cause confusion for students while including it in the language has been seen to cause confusion and more errors.

Methods may neither be overloaded, in accordance with Observation 2.6, nor declared `void`. In a language without mutation, void methods can have little visible affect, and so would not serve a purpose in the language. Further, the material presented while using this language level requires students to produce a testable value for each method they write. This reasoning extends to require that each `return` statement returns a value.

For the statements and expressions, presented in Table 2.2, Beginner imposes restrictions to avoid the errors discussed in Section 2.2. In accordance with Observations 2.3, 2.2, and 2.9, `if` requires `else`, field and method names cannot overlap, and some library methods from `Object` (such as clone) cannot be called.

Additionally, the overloaded portion of `+` to support String concatenation has been

**Table 2.2**. ProfessorJ Beginner Statements and Expressions

| Construct | Restrictions |
|---|---|
| Statements | |
| if | Must have else |
| return | Must have expression |
| assignment | Cannot be +=,-=, etc.<br>Only in constructor |
| Excluded statements:<br>block of statements, variable declaration, throw, while, do,<br>for, try, switch, break, continue, label, synchronized, ++ and -- | |
| Expressions | |
| Literals | (excluding null) |
| this | |
| Binary operations | + may not be used as string append |
| Unary operations | ++, -- not allowed |
| Variable reference | |
| Field access | `this` required to access current object's fields |
| method call | `this` required to call current object's methods |
| class allocation | |
| Excluded expressions:<br>cast, qualified name access, array access,<br>array allocation, array instantiation, ? conditional,<br>instanceof, assignment | |

removed, as this operation often causes students confusion, and removing the shorthand does not remove any functionality.

Field and method accesses on the current object must be prefaced with `this` instead of using the implicit `this` provided by Java. This requirement stems from observations made by other curriculum developers who noted that students understood object-oriented programming concepts better with this restriction. Coincidentally, this restriction also permits better error messages to distinguish between field accesses and method parameters, as well as in field initializations within the constructor.

Within this language, the `else` keyword requirement may not be technically necessary, since without it a program such as

```
if (this.condition())
  return 1;
return 2;
```

will produce `1` only when the condition is `true` and `2` only when it is `false`. However, the Intermediate language needs to require the `else` keyword to prevent the observed problems. Pedagogic languages should be subsets of each other, so that students are not required to unlearn semantics covered in the previous languages. Therefore, to allow Intermediate to restrict the `if` statement, Beginner must also require the `else` keyword.

As previously mentioned, the language levels have evolved to accommodate new observations and curricular changes. Originally, Beginner contained classes and abstract classes instead of interfaces. Fields were implicitly private to begin teaching encapsulation early. As the curriculum changed to present these topics later, the language levels also changed to support the new direction of the course.

**2.4.1.1 Ongoing observation.** Presently, the lack of local variables within Beginner causes problems for students and teachers. Students quickly reach a point where they are not ready to graduate to the Intermediate language level, yet are writing programs with sufficient complexity to warrant the use of a local variable to properly abstract their code. However, extending Beginner to allow local variables opens problems for early programmers who do not properly understand when to use a local variable. This conflict suggests that the step between Beginner and Intermediate may be too large in practice, and another language level including local variables and perhaps other features of Intermediate may provide better support.

### 2.4.2 Intermediate

The ProfessorJ Intermediate level supports the curriculum from the introduction of inheritance until the introduction of overloading. This includes fully extendible classes, interfaces, overrideable methods, and mutable fields. Table 2.3 presents the declaration constructs of this language level. Classes and methods may be declared abstract, but no other modifiers are supported.

With the addition of abstract classes, students learn to avoid repetitious code between subclasses by lifting methods into their abstract class. This teaches them a key difference between interfaces and abstract classes. This level also supports the introduction of implementing multiple interfaces and employing object-polymorphism. Therefore, expressions such as `instanceof` and `casts` are now required, as shown in Table 2.4.

To support mutating field values, the possible statements are extended with blocks and assignment. However, based on Observation 2.10 using assignment as an expression is still prohibited. Within Intermediate, fields may not be declared `final`, as the curriculum presents this concept later in the course. This restriction may appear incongruous with the Beginner language, where fields default to `final`. However, since the Beginner language does not allow the mutation of field values, the `final` property of the fields is technically unnecessary and undetectable to students.

Within the Beginner level, constructors could contain no statements other than as-

**Table 2.3**. ProfessorJ Intermediate Declaration Constructs

| Construct | Restrictions |
|---|---|
| imports | Some imports not allowed |
| classes | Implicitly public<br>Cannot be final<br>Cannot have static members |
| interfaces | Implicitly public<br>Cannot have static members |
| fields | Implicitly public<br>Cannot be final |
| constructors | Implicitly public<br>Cannot be overloaded |
| methods | Implicitly public<br>Cannot be overloaded |
| Exclusions:<br>package, class and instance initialization blocks, inner classes and interfaces, field and method modifiers, arrays | |

**Table 2.4**. Intermediate Java Statements and Expressions

| Construct | Restrictions |
|---|---|
| Statements | |
| if | Must have else |
| return | |
| block of Statements | |
| assignment | Cannot be +=,-=, etc. |
| method call | |
| variable declaration | |
| Excluded statements: throw, while, do, for, try, switch, break, continue, label, synchronized, ++ and -- | |
| Expressions | |
| Literals | |
| this | |
| Binary operations | + may not be used as string append |
| Unary operations | ++, -- not allowed |
| Variable reference | |
| Field access | |
| method call | |
| class allocation | |
| cast | |
| instanceof | |
| Excluded expressions: qualified name access, array access, array allocation, array instantiation, ? conditional, assignment | |

signments to initialize fields. Intermediate introduces the idea that a constructor can do additional work, and so the body of a constructor has no restrictions beyond that of a method returning no value.

Expressions and statements retain many of the restrictions from the Beginner level to avoid the same problems. As Intermediate introduces inheritance, it supports the restrictions discussed in Observations 2.7 and 2.11 to avoid these problems when introducing other concepts of inheritance.

**2.4.2.1 Ongoing observation.** The current version of Intermediate does not require that all fields be initialized either at their declaration point or within the constructor (in order to create cyclic-data references). This permits the common problems arising from the `null` value, namely runtime errors reporting null pointer accesses.

Presently, the compiler does not prohibit this situation and attempts to alleviate any

confusions by providing a student-targeted runtime error. However, observations are indicating that this support is insufficient. Therefore, future versions of the Intermediate language may require all fields to be initialized as in the Beginner language, where `null` is a possible value to use.

### 2.4.3 Advanced

The ProfessorJ Advanced language covers the curriculum from the introduction of overloading and access modifiers through the presentation of iterative programming. After this topic, students graduate to the full Java language.

As this language level is used to present library development and use, it must support packages, access modifiers and static members. Also, to support iterative programming, it must include arrays, loops and supporting expressions. Table 2.5 shows the constructs supported in Advanced. By this point, students are largely familiar with the Java language, and most restrictions have been lifted.

The primary omissions between Advanced and full Java for declaration constructs are inner classes. Presently, the curriculum designers (including myself) are considering introducing inner classes to Advanced. Both the structure of the course and the ability to properly support the construct with error messages are contributing to the decision. While students are no longer likely to accidentally create an inner class, these constructs introduce other difficulties including accessing members of the enclosing class, the difference between `static` and non-`static` nested classes, and other errors regarding access.

In addition to this restriction, fields and classes cannot be declared `final` whereas

**Table 2.5**. ProfessorJ Advanced Declaration Constructs

| Construct | Restrictions |
|---|---|
| package | |
| imports | Some imports not allowed |
| classes | Cannot be final |
| interfaces | |
| arrays | |
| fields | Cannot be final |
| constructors | |
| methods | |
| class initializers | |
| Exclusions:<br>class initialization blocks, inner classes and interfaces | |

methods can. Final method support was desired to explain certain libraries, whereas final fields and classes are explained using full Java. Additionally, as threading and synchronization are not taught within the second-semester course, the `synchronized` modifier is not included within the language.

Table 2.6 presents the statements and expressions supported within the Advanced language.

While students using the Intermediate language begin to augment their programs with external libraries, students using Advanced begin to create their own and learn to fully

**Table 2.6**. ProfessorJ Advanced Statements and Expressions

| Construct | Restrictions |
|---|---|
| Statements | |
| if | |
| return | |
| block of Statements | |
| assignment | |
| method call | |
| variable declaration | |
| while | |
| for | |
| do | |
| break | inside a loop |
| continue | inside a loop |
| Unary operations ++ & -- | |
| Excluded statements: throw, try, switch, label, synchronized | |
| Expressions | |
| Literals | |
| this | |
| Binary operations | |
| Unary operations | |
| Variable reference | |
| Field access | |
| Array access | |
| method call | |
| class allocation | |
| array allocation | |
| array initialization | Cannot be anonymous |
| cast | |
| instanceof | |
| ? conditional | |
| Excluded expressions: qualified name access, assignment | |

interact with the Java API, and other APIs. Therefore, students need to understand and gain experience with `public`, `private`, and `protected`, as well as `package`s and package level access.

Further, students are introduced to the differences between class specific members and instance specific, and learn about abstraction through overloading constructors and methods. At this point, students split multiple classes over multiple files, so names can be prefixed with the package containing them.

Notably, despite students accessing and reading external libraries, Advanced does not support exception handling statements, `throw` expressions, or `throws` declarations, due to curriculum decisions. Nevertheless, programs may call methods that throw non-`RuntimeException` derived exceptions. The compiler treats all exceptions arising in library code identically to `RuntimeException` methods, so that students do not need to understand exceptions to interact with these libraries.

Similarly, the `break` and `continue` statements outside of the context of looping constructs are not covered before students graduate to full Java, and are therefore not permitted anywhere else within the program. While Section 2.4.1 discussed not restricting language constructs based on context to prevent confusion, this restriction of `break` and `continue` does not contradict the earlier statement. Within Java, these statements only have meaning within loops, `switch` statements, and labeled statements. Of these three, only loops may occur within Advanced programs.

Other statements that facilitate programs with loops, such as `++` and `--`, also fit into the material at the same point. These expressions are necessary for `for` loops unless the restriction on assignment as an expression is lifted. Current investigations suggest that this restriction may indeed no longer be beneficial to students; however, further investigation is required before removing it.

Anonymous array initialization is not necessary for programs at this level; the array can always be given a name and then accessed. Teaching this syntactic short-hand is not planned, so the construct is not permitted in the language.

Within Table 2.6, there are no restrictions placed on the `if` statement. This means that programs may now omit the `else` branch. Observations have indicated that students using Advanced have enough practice with both the statement and blocks of statements by this time to understand the significance of omitting the `else` clause, and some novice

students begin to complain about the uselessness of `else` in certain situations.[1] Therefore, the clause can be omitted.

## 2.5 Connecting to the Environment

Program development environments provide novices and professionals greater support than editing the program, invoking the compiler, and running the program using two or three separate programs. Different pedagogical and professional environments provide additional features and connections, including interactive access to the program, integrated debugging, visualization tools, automated testing, etc. The features of the environment affect which language constructs are necessary.

The ProfessorJ languages reside within the DrScheme development environment, which is a pedagogic/professional environment that supports several languages. DrScheme presents two windows to users, an upper window referred to as the definitions window and a lower window referred to as the interactions window. Full programs are created within the definitions window. Experiments on the program occur in the interactions window; in full read-eval-print-loop style. Figure 2.1 shows a sample session of the ProfessorJ Beginner language within DrScheme.

When in the ProfessorJ language levels, interfaces and classes may be declared in the definitions window. Essentially the content of the definitions window corresponds to the contents of a typical `.java` file, except that for the first two language levels, multiple classes are the norm instead of the exception. Classes and interfaces cannot appear within the interactions window, instead statements and expressions may appear there.

A typical session begins by implementing a syntactically complete class in the definitions window. To compile and access the class, the programmer selects DrScheme's Run button. If there are any errors, the error is highlighted in the definitions window and the error message appears in the interactions window. If there are no errors, the class can now be used within the interactions window. Instances of classes display in a format that includes the name and assigned values of all the fields in the class.

Once students start writing methods, they need a mechanism for (repeatably) testing the implementation of these methods for each class. Static methods typically serve roles in testing, either through using `main` or using jUnit. Despite these roles, my observations indicate that static methods cause more problems than they solve and allowing static

[1] The removal of this restriction does need specific class attention to point out potential errors.

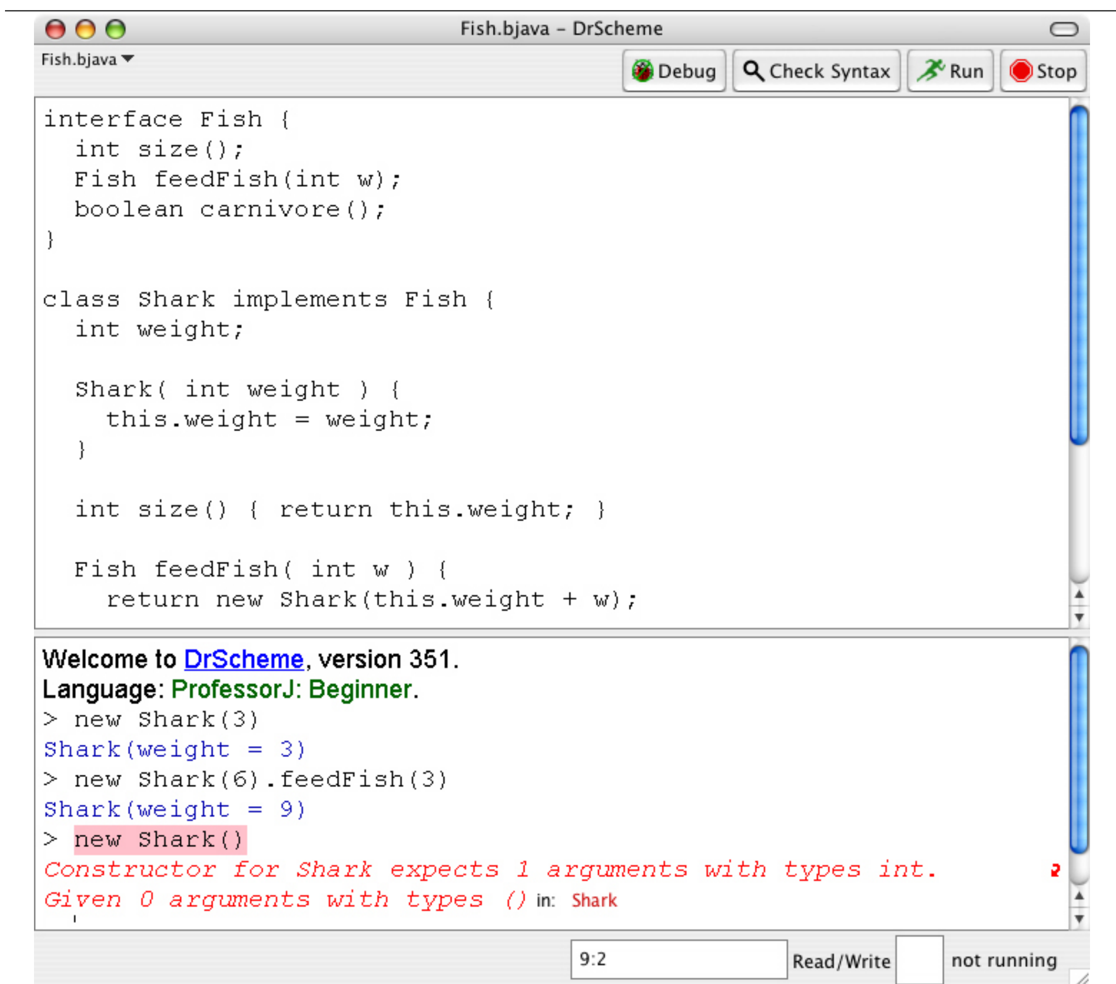


**Figure 2.1**. ProfessorJ Beginner program within DrScheme

members for the sole purpose of testing violates the intentions of the early language levels. Another solution is needed.

An extension to the DrScheme environment supports placing test expressions in graphical boxes. These can be placed in the definitions window, allowing savable, repeatable tests within the confines of the language level restrictions. The boxes visually indicate (with a red $*$ or a green check) whether the test passed or failed.

However, experience with these boxes indicates that they are not an ideal testing mechanism as programs become more complex. Additionally, while some students find the graphical boxes intuitive and enjoyable, others find the user interface cumbersome. Further, they do not lead to a permanent testing strategy for students, which may cause the students to cease testing once boxes are no longer available.

Another testing solution uses an Example class within the definitions window. Students store instances of their other classes within this class and write boolean fields to store all of the tests of the methods. This solution fits within the language levels and scales to more complicated programs, as well as transitioning the students to using a professional testing tool such as jUnit. However, this solution requires that students write methods to compare two instances of the object sooner than the curriculum intends. Also, the true or false values of the variable are only visible when the student instantiates the Example class in the interactions window, so a nondiligent student might not know which change caused an existing test to fail.

Work on a testing solution that combines features of both these systems and leads students towards writing tests with jUnit is ongoing.

## 2.6 Supporting Error Messages

Restricting a professional programming language removes potentially erroneous situations, which allows a compiler to narrow down the possible errors in a situation. This in turn allows the compiler to produce a more targeted error message than would make sense in a more syntactically rich language. For novice programmers, these targeted error messages reduce the confusion involved in correcting mistakes.

In general, error messages in a professional compiler do not adequately support students due to their vocabulary and tendency to terseness. Professional programmers rarely need to read an error message beyond the source of the error, so the content of the message attempts to convey the essence of the problem with minimal effort for the programmer to read and the compiler writer to create. Students can be taught to read error messages when they provide sufficient information to explain and identify the problem, so pedagogic error messages can be more verbose to assist the student.

The remainder of this section presents a common set of mistakes that are not solved by restricting the programming language. Providing explanatory error messages in these situations allows students to improve their understanding of the programming language and programming techniques without the intervention of a teaching assistant.

One class of errors that tend to cause confusion are the result of typos. Leaving out the letter of a keyword, swapping two characters, or capitalizing some, cause an error message for most compilers that does not indicate the found identifier is close to the desired one. Many people when reading a word that is close to the correct spelling will

tend to read it correctly despite the error. Therefore, it can be difficult to properly trace the error. Adding in a small keyword spell checker demonstrates this error, so that a misspelling of `public` as `pulic` will report that the compiler has encountered a word similar to keyword `public` that is misspelled.

As previously mentioned, students are observably confused with error messages reporting unbound identifiers when they can clearly see the binding of the identifier. One circumstance of this involves a program similar to

```
interface Animal {
  boolean vertebrate();
}
class Cow implements Animal {
  boolean vertebrate() { return true; }
  boolean vegetarian() { return true; }
}
```

and a call-site similar to

```
Animal m = new Cow();
m.vegetarian()
```

Typical error messages indicate that the method `vegetarian` is unknown, which confuses students who are doing constant propagation in their heads. ProfessorJ attempts to lower this confusion by explicitly stating the type of `m` and indicating that it is `Animal` that lacks the method, to try to direct students to the source of the error.

Similarly, overloading error messages continue to confuse students even when overloading appears in the language. Therefore, ProfessorJ provides overloading error messages specialized to three different situations with method lookup errors in the presence of overloading. If there are no methods with the base name, the resulting error indicates this specific information with no mention of the called method's arity or argument types. If there are methods with the given name but not the given arity, the error message indicates that "No definition of Method with N argument(s) was found." And finally, if the arity hits a match but the types of the arguments are incompatible, the given types and allowable types are outlined in the error message. This added information narrows the students' focus onto the core of their error, instead of lumping the three possible areas of inconsistencies into one general error message.

## 2.7 Related Work

Work on pedagogical alternatives to professional tools has encompassed many different languages and ideas over the years. Prior tools supported pedagogic languages based on restrictions of professional languages. Other tools attempt to minimize student problems with new pedagogic languages or with entirely different forms of support from the development environment using a full professional language.

### 2.7.1 Language Levels

In the mid 70s, Holt et al. created a series of pedagogic subsets based on PL/I called SP/k [1]. Like ProfessorJ, each subset presents the language constructs that support a portion of the curriculum, and each builds on the previous subset. Also using PL/I, Conway and Constable created one subset, PL/CS [8] to use in an introductory course without the added complexity of the full language. Both of these systems targeted their error messages for the students knowledge level.

The Cornell Program Synthesizer [9] later provided an integrated editing environment for PL/CS that prevented syntax mistakes for students with structured editing. The environment also allowed students to interact with their programs during development.

Also in the 70s, Wirth created a compiler and interpreter for a subset of Pascal, Pascal-S [10], with a similar philosophy. Again, constructs used within the introductory course were supported and all others omitted. Error messages were targeted to students. Like PL/CS, with only one language for the entire first course, students could still access constructs significantly before their introduction.

In 1993, Ruckert and Halpern [11] presented a subset of the C language which restricted the static type system to prevent common student errors, including adding booleans and removing potentially unsafe type casts. Students use only the one subset throughout the course. Additionally, educationalC did not allow students to access external libraries or teacher provided libraries.

The DrScheme development environment [2] began as a development environment for three subsets of the PLT Scheme language. These languages support students using the "How to Design Programs"[6] curriculum and provide crafted error messages designed for students. As previously mentioned, the ProfessorJ languages support a follow-up curriculum to "How to Design Programs" and also reside in the DrScheme environment. The first curriculum covers functional programming in a dynamic language with algebraic

data-types, while the ProfessorJ languages bring in object-oriented programming in a statically typed language.

Like DrScheme, the DrJava [4] environment provides a definitions and interactions window to aide student development. After compiling, programs written in the definitions window are accessible in the interactions window. DrJava also supports connections to jUnit and focuses on professional level testing from the first.

In 2005, the DrJava team presented Java language levels to support the curriculum accompanying DrJava [12]. As in earlier language level implementations, error messages are geared towards students and the curriculum. Additionally, while the initial DrJava subset does not support static members or value-less methods, jUnit testing still occurs in this subset. Therefore, the language within a test class is different from that in other classes.

The differences between the ProfessorJ and DrJava language subsets highlight the need for language levels to be tailored to specific curriculums. For example, the ProfessorJ curriculum stresses the importance of students understanding the connections between constructor arguments and field values, as a transition away from algebraic data structures. The DrJava curriculum does not stress this connection and automatically creates a constructor for students. Further, the DrJava curriculum moves to library interactions much earlier than the curriculum with ProfessorJ, necessitating various library support constructs in the second language. Curriculums with these different requirements cannot share language subsets, necessitating different systems. With a declarative language specification, both curriculums could support their language subsets without requiring two full type-checking implementations for the two systems.

Concurrent with the ProfessorJ development, the Expresso [13] project introduced a compiler with student-oriented messages. This system noted common student errors, such as omitting the `else` clause from an `if` statement, and searched the submitted program for these forms of errors. Additionally, as discussed in Section 2.6, the overall language of error messages attempts to address a students knowledge level. Overall, however, this system does not provide many language restrictions and novice students may still accidentally write syntactically valid but technically incorrect programs that are difficult for them to understand and debug.

### 2.7.2 Pedagogic Languages

Many different languages have been created for pedagogic use, instead of using professional languages. These include both graphical languages that merge the language and the environment, and textual languages that either operate without an environment or do not merge the language and environment.

Textual languages, including Jeroo [14] and Logo [15], may be inspired by professional languages and can be used to teach several styles of programming. Due to the control over syntax and semantics, error messages can be more targeted than error messages for pedagogic subsets. For example, one very difficult error situation within ProfessorJ occurs when three identifiers appear in a row, such as `Fish eat Food`, this sequence is either the incorrect opening for a method or a field, the compiler cannot distinguish and so cannot provide a precise error message. In Jeroo, methods begin with a special keyword, so that this situation is avoided.

Graphical languages, including Alice [16], toontalk [17], and jPie [18], remove the burden of syntactic errors from students by supporting varying degrees of structured editing. These tools may also connect the program to a visual world to run and interact with program elements, like Alice, or may also be inspired by professional languages, like jPie.

Many of these tools are not intended for large programming projects, and just introduce students to the ideas and concepts of programming. These students many encounter the same problems as a novice when facing a professional compiler and the full syntax of a professional language all at once. Further, of these tools, only jPie connects to the professional libraries, so instructors must create their own libraries for the other tools.

### 2.7.3 Pedagogic Tools Without Language Levels

Other tools provide a pedagogic interface over a professional language that is oriented towards students. These systems expose students to the full error messages of the professional language. These systems support curriculums ranging from middle school students to introductory college students.

For example, the Squeak system [19] is used to introduce middle school students to both programming and scientific thinking, using the Etoys interface. Squeak is a graphical system in which the underlying objects are in the smalltalk language. Students program in the Etoys interface by manipulating prepared graphical widgets and directing objects as to how to move and respond to their environment. While the Etoys interface

to the Squeak system is intended for students, the system operates directly on top of the professional environment Squeak. This provides the flexibility to create large scale projects. It also, however, allows students to encounter the error messages and constructs of the full system, which can add unnecessary confusion.

Among the development environments for full Java used in schools, BlueJ [7] is the primary environment designed for student instruction. BlueJ focuses on teaching introductory students object-oriented programming while using the environment to shield students from some of the languages complexities. To focus on OO programming, BlueJ graphically presents a collection of classes, with arrows denoting the inheritance and use relationships between them. Students can create classes by adding boxes in the environment, and they can build inheritance hierarchies by inserting arrows. Creating a class in this manner generates a bare bones class in the source code, with the class name and inheritance specified, as well as an example method and field. The method demonstrates the syntax for a method declaration (return type, name, and modifiers) and a return statement.

Once a class has been implemented and compiled, students can interactively create new instances, dispatch public methods and inspect fields. This functionality facilitates interactive testing(without writing specific test code or requiring an understanding of static methods), as well as visually presenting the inheritance of methods and fields. BlueJ also contains a statement and expression evaluator, where students can test code. BlueJ provides access to a graphical debugger, which sets breakpoints and allows users to step through execution (*a la* gdb).

Used only as an editor and interactive environment, BlueJ partially succeeds in shielding students from static members. Students do not have to write statics for their programs to work. However, because BlueJ does not have language levels enforcing this protection, students may still use static members (often incorrectly). Additionally, if students use the debugger, they are exposed to the concept of statics (and threads) before the material is presented in their course. A further problem with BlueJ, as has been discussed above, is that the error messages provided to students do not target their knowledge level. This problem has been mentioned in a user study [20] as one of the largest problems with BlueJ. The graphical view provided by BlueJ is beneficial to students as they learn to design and reason about object-oriented programming.

# CHAPTER 3

# ADDING LIBRARY SUPPORT

Encouraging students to program with external libraries combines practical education with more enjoyable programming. Instead of writing small stand-alone programs, students can create or extend games, develop interactive web pages, write weather forecasting engines, etc. Such projects provide students with experience interacting with others' implementations as well as a finished project that they can use and tend to enjoy.

Most libraries are intended for professional programmers, so they come with many problems similar to those of professional languages and compilers — including inappropriate error messages and features beyond the students' experience. In addition, the interface to a library may not fit into the language expectations of the current pedagogic subset. In particular, a library may require extending an abstract class, where a pedagogic subset may not support inheritance. For these reasons, instructors may need to provide teaching libraries to connect the students' programs with the professional libraries.

A teaching library serves as a bridge between the professional language of the full library and the pedagogic language of the students' programs. When the static-semantics of the pedagogic and full languages match, the teaching library bridges the difference between the students' knowledge level and the professional interface, for example removing the need to implement an interface if the pedagogic language does not contain interfaces. Data checks may be necessary in these teaching libraries to avoid runtime errors from the professional library, which may use inappropriate vocabulary. In cases where the static-semantics of the two languages do not match, the teaching library must dynamically merge the static semantics by checking, marshaling, and protecting values as they pass between the two parts of the program.

Type checks and conversions fit into one of two categories, in-line checks or wrappers. In-line type checks and conversions resemble the data checks necessary to provide student-oriented error messages. These checks can immediately verify that the received datum meets the type requirements and can be immediately converted into the proper

representation. In-line checks are not sufficient for all libraries, however, particularly not for libraries that pass objects between the student's program and the professional library. In these transactions, a wrapper surrounds the object, performing all the necessary checks and conversions around accesses before dispatching to the object.

Adding the wrappers (and in-line checks) causes significant programming overhead to developing teaching libraries, and it adds potential failure points in the program when checks are incorrect or omitted. With multiple pedagogic languages to bridge with a professional library, the instructor's task becomes daunting.

By adding these type-checks and data conversions, the library implementor is actually writing a specialized interface allowing two languages to interoperate. The difficulty involved matches the general difficulty of writing programs in multiple programming languages. By extending a programming language to support interoperability, the task is simplified, despite bridging multiple incompatibilities between the various languages directly.

This chapter first presents some illustrative examples of mixing programs in pedagogic languages with programs in other languages. It then presents the language extension (`dynamic`) designed to support fine-grained interoperability, followed by an examination of using the language feature to rework the examples. The chapter concludes with an extended example of a pedagogic library using both the `dynamic` language extension and using older technologies.

## 3.1 Language Mixings

Different pedagogic languages can arise from different course needs, as mentioned in Chapter 2. This section presents example combinations between pedagogic and professional languages in the context of creating a teaching library to demonstrate the difficulty of this task without fine-grained interoperability.

### 3.1.1 A Dynamically-Checked Pedagogic Language

One potential pedagogic subset of Java removes all static type information from the language. Any teaching library connecting a program in this subset to a library in full Java must perform dynamic checks to ensure that the data conforms to the type expectations of the library.

The program in Figure 3.1(a) could be written by a student using a dynamically-checked syntactic Java subset. In order to provide a library to draw trees for the student,

```
class Tree {
  height; age; fruit;
  averageGrowth() { return height/age; }
}
```

(a) Student Program

```
void printTree(Object tree) {
  ... try { (Float) tree.getClass().getField("height").get(e) ... }
  catch (ClassCastException e) { ... }
  ... try { ... (Integer) tree.getClass().getField("age").get(e) ... }
  catch (ClassCastException e) { ... }
  ... printFruit(tree.getClass().getField("fruit").get(e)) ... }
}
```

(b) Teaching Library

**Figure 3.1**. Combining dynamic pedagogic language with Java

an implementor using standard technology might write the `printTree` method seen in Figure 3.1(b).

This program uses reflection methods of the student's `Tree` object to access its values, a traditional means of supporting interoperability. Once retrieved, the format of the value must be checked through casts (which also should be caught and handled with informative error messages). Programming in this style is cumbersome and error-prone.

Problems compound in this system when an object from the student's program needs to be used as a statically known class. In these circumstances, the value must be wrapped with a subclass of the known class and the wrapper must implement checks on the returned values of all of the methods within the dynamically-checked class.

### 3.1.2 Parametrically Polymorphic Library

The latest Java language version includes parametric polymorphism, but a pedagogic language might exclude this feature. When programming in the pedagogic language, students should not encounter error messages referring to incorrect uses of polymorphic entities.

Useful libraries may incorporate generic specifications within their interfaces, requiring the creation of a teaching library to modify the interface for students using a language without generics. The portion of a GUI library in Figure 3.2(a) demonstrates a potential use of generics within a library. The methods to manipulate the `Frame`'s `Panel`s require that the `Panel`s be wrapped in a `List`. While the student program (Figure 3.2(c)) can

```
package widgets;

class Frame {
   List<Panel> getPanels() ...
   void  installPanels( List<Panel> l) ...
}
```

(a) Library using a polymorphic List

```
class Frame extends widgets.Frame {
  void installPanels( List l ) {
    for( ... )
      if (! l.entry(i) Panel)
         throw ...
    super.installPanels(l);
  }
}
```

(b) Teaching library removing polymorphism

```
class SimpleGUI {
  Frame mainWindow;
  List subWindows;

  void display() {
    subWindows = subWindows.add(new Panel(...));
    ...
    mainWindow.installPanels( subWindows );
  }
}
```

(c) Student program using Frame

**Figure 3.2**. Combining polymorphic library with nonpolymorphic language

construct a `List` of `Panel`s, the language does not support the `List<Panel>` type. The teaching library bridges this gap.

The teaching library, Figure 3.2(b), overrides the `Frame` class to provide a method with an appropriate type. The new method checks that all members of the provided `List` are `Panel`s before passing the value to the super class. To allow students to use this library, warnings regarding potential conflicts in the polymorphic types must be suppressed. Alternately, the teaching library could repackage the `Panel` values into a new `List<Panel>` before passing them to the super method, which would avoid the suppression of warnings but requires the creation of an additional List.

### 3.1.3 A Dynamically-Typed Library

While the previous sections discussed pedagogic languages with weaker type systems than the professional library, the reverse situation can also occur. In fact, my original motivation for fine-grained interoperability was to allow the Java pedagogic subsets to connect to Scheme libraries, such as graphics libraries. As I will show, `dynamic` offers a general solution for connecting two languages with different static semantics.

Traditional techniques for this bridge require either access through an API, as seen in Figure 1.1(a), or through the Java native interface (another form of interoperability API). Both situations require that the programmer insert checks and conversions on values that return from Scheme and insert guards on objects that are passed to Scheme functions. Further, calls into Scheme are obfuscated by the API middleman. Section 3.4 presents a complete implementation using the native interface to compare with the fine-grained solution.

## 3.2 Extending the Language with `dynamic`

By extending the language with dynamically-checked expressions, interoperability between languages with different static-semantics becomes easier for the programmer. Specifically, adding a dynamic type annotation and specialized import statement sufficiently inform the compiler where to insert checks so that the programmer can safely omit all type-checks.

### 3.2.1 Syntax and Extensions

Within my extended language, variables can be declared to have `dynamic` type.

```
class Printer {
  dynamic tree;
  printTree( dynamic tree ) ...
}
```

Declaring a variable with the `dynamic` type specifies that the value assigned to this variable will be checked, and converted where necessary, during execution. No usage of any value with the `dynamic` type can generate a compile-time error. This capability allows `dynamic` values to bridge two languages with different type-expectations.

Internally, all variables imported from libraries written in other languages have the `dynamic` type. A small extension to the Java import declaration indicates when the

imported library is not a Java program. When the import declaration begins with the word `scheme`, compiler imports dynamically typed libraries from the Scheme language.

```
import scheme.graphics;
import scheme.lib.prettyPrint;
```

Although the `dynamic` extension is not specific to Scheme, I use `scheme` as a place holder for "implementation language of the dynamically typed library". The language created by merging these extensions with the Java programming language is referred to as Java + `dynamic` .

The preceding import statements direct the compiler to accept values from a Scheme graphics library in the current package and from a Scheme *pretty-print* library in the main repository for Scheme libraries. Unlike other Java classes, Scheme libraries contained in the current package are not automatically visible to the Java library. Programs written in the subsets of Java are imported using the standard Java import.

Syntactically, values from Scheme libraries appear as static members of a class with the name of the Scheme library. In a program with the above imports, the code to access a print function from the pretty-print library is `prettyPrint.print`. The compiler statically guarantees that the pretty-print library contains and exposes a value named print, but other static checks occur for Scheme imports.

Within Java, values may be flat data or objects. Within Scheme, values may also be functions. Therefore, values with the type dynamic may also be used as functions in Java. Within the following example, a parameter to a Java method is accessed as a function:

```
String keyCallback(dynamic callback, Charset c) {
  return callback(c);
}
```

Assuming that the first argument to `keyCallback` is a function from Charset to strings, this program raises no errors.

Another important role of `dynamic` is its effect on method calls and other primitive operations. Consider the following example.

```
Fruit getFruit( dynamic tree ) {
  if (tree.hasFruit())
     return tree.pickFruit();
  else
     return tree.getSeeds();
}
```

The `tree` variable is used as an object with no declared class. Nevertheless, methods and fields can still be accessed from `tree`. These fields and methods need not come from any known class.

Consider the first use of `tree`, which accesses a `hasFruit` method of no arguments returning a boolean. During execution, the value is inspected to ensure that it is an object. This object is further inspected to ensure that it contains the correct method, which is then called. Separate checks will be used for calls to `pickFruit` and `getSeeds`. Indeed, no class with all three methods may exist, which is completely acceptable within this program.

Chapter 4 discusses the details of the Java + `dynamic` language, and the remainder of this chapter discusses how to use `dynamic` in creating teaching libraries.

## 3.3 Writing Libraries with `dynamic`

Section 3.1 presents possible teaching languages/professional library pairings. In those examples, the teaching libraries bridged the language gap using traditional interoperability technology. This section presents teaching libraries that use the Java + `dynamic` language to bridge the same programs/languages.

### 3.3.1 Dynamically-Checked Student Language

The teaching library in Section 3.1.1 combined a dynamically-checked student language with the static type checking of Java. The library used reflection and casts to access and verify the values from the student's program.

In an updated version of the program, these extra calls are removed, leaving

```
void printTree(dynamic tree) {
  ... tree.height / 3.0 ...
  ... tree.age < 100 ...
  ... printFruit( tree.fruit ) ...
}
```

Internally, in this revised code, similar checks to the hand-implemented ones ensure that the `tree` object has the desired fields, and that the values of these fields conform to the requirements of their uses.

In this library, `printFruit` can either accept an argument of type `dynamic` or one of type `Fruit` with the same code for `printTree`. In the latter case, `tree.fruit` must match the signature for a `Fruit` class, the `tree.fruit` is inserted into a wrapper that checks the types of fields and methods on access.

If the value for `tree` given to `printTree` is not an object with one of the required fields, an error will occur during execution. For example, if the `tree` value does not contain an age field, the error message indicates that within the class containing `printTree`, at the `tree.age` position, the object was required to have a field with name `age` but did not. Further, if the age is not an integral number, the error report indicates that an integer is required.

The resulting behavior of the system matches that of the initial bridge library, which directly required the programmer to insert the runtime checks, without requiring as much overhead and potential for error.

### 3.3.2 Polymorphic Library

With the example from Section 3.1.2, a teaching library bridges two typed languages with differing levels of strictness. The professional library allows parametric polymorphism and adds static guarantees to ensure that certain conditions are met. The pedagogic language does not support type constraints.

Within the extended system, the teaching language bridges the two languages by using a `dynamic` parameter to indicate the point where the two type systems do not match.

```
class Frame extends widgets.Frame {
  void installPanels( dynamic l ) {
    super.installPanels(l);
  }
}
```

This allows the value to enter the original frame without static conversions or checks of the data. These checks are still necessary, and will be shifted to run-time checks within the accesses to the list from the students programs. The wrapped list will also carry with it the source location of the extending Frame class so that errors are correctly reported as stemming from this location.

## 3.4 A Teaching Library

The primary use of the Java + dynamic language extension at present is to connect the Scheme graphics libraries to the pedagogic languages of ProfessorJ through a teaching library. This library requires interoperation between Java and Scheme. In particular, the connecting library should be written in Java, so that students programming in the Java-esque subsets have a firm Java interface to program against. Conforming to Java

interfaces comprises one facet of the students education, so the types of the library should be expressible in standard Java.

### 3.4.1 The Library

The teaching library allows students to create and manipulate images and to program animations and games. A student writes the logic and control of a game, or the code to manipulate images. The teaching library connects to an existing pedagogic graphics library and the Scheme MrEd graphical system [21] to provide bit-level drawing support and window management.

The teaching library provides several classes, including color, position, and drawing command representations. The primary support of the library is contained in three classes: Image, View, and World (and its implementing subclasses, notably GameWorld). See Figure 3.3 for the visible interface of these classes.

The Image class allows students to represent and manipulate pictures imported from JPEGs, BMPs, etc. This class connects to an existing Scheme library, providing no additional functionality.

The View class allows students to open a window and draw Images into it. This class connects to various components of the Scheme windowing system, merging the interactions into one interface.

The World family of classes both guide the student into a programming pattern for creating their animated/interactive programs and provide the support for interacting with the visible canvas. The class uses existing Scheme libraries to provide time and key events to the student's program.

### 3.4.2 Implementation Options

The teaching library may be implemented using a few different technologies to connect the languages: through an interoperability or reflection API, through the Java Native Interface, or by using Java + `dynamic` .

The first option requires the creation of a library to generally connect Java and Scheme, similar to those created by other Scheme implementations that compile to Java such as jScheme[22]. As the ProfessorJ system operates by compiling Java into Scheme (see Chapter 5 for details of compilation) and can easily use the Java native interface, such a library has not been created.

```
class Image {
 Image overlay( Image i);
 boolean inside( Image i );
 //Several other Image manipulation methods
}

class View {
  View();
  View display(int width, int height);
  View hide();
  View show();
  Image draw(Command c);
  Image drawSequence( CommandSequence cs);
}

abstract class World {
  View display;
  World onKey(String key);
  World onTick();
  Image draw();
  Image erase();
  boolean animate( int width, int height, int rate );
}
```

**Figure 3.3**. Drawing library: Student interface

Within the ProfessorJ system, native methods are implemented in Scheme. The compilation procedure specifies a name for these methods, which appear as Scheme functions within another file. Each (nonstatic) native function is automatically given several data-structures containing information about the current instance. The Scheme functions are unrestricted by the Java compilation. In addition to providing access to Scheme functions through the native interface, the ProfessorJ system also allows Scheme classes to impersonate Java classes by providing a specification of the field and method interfaces of the class, i.e., the Java type signatures for the Scheme class. These assertions are unchecked.

The ProfessorJ system also supports the Java + `dynamic` language option. Programs written in Java + `dynamic` have access within the Java program to all of the PLT Scheme libraries, as well as locally written Scheme modules. In some circumstances, the Java programmer will have to write an auxiliary Scheme module to translate some Scheme

names into Java or to write functions.

## 3.5 The Implementations

This section presents illustrative portions of the native and Java + dynamic implementations of the graphics library. The complete implementations of these classes are contained in Appendix A.

### 3.5.1 Image

The `Image` class embeds a Scheme value, representing a picture, and provides methods to manipulate this value. The methods connect to functions provided by a Scheme *image* library.

This section presents two representative methods: `inside`, which determines whether the given `Image` occurs within `this`; and `addLine`, which uses two points, represented by Java class `Posn`, and a `Color` to create a new `Image` containing a line.

**3.5.1.1 Native Method Implementation.** Figure 3.4(a) presents the Java portion of the native method implementation. The `Image` class stores the Scheme value as a `private Object` field, `theImage`. While the value assigned to this variable will never have type `Object`, no Java code will ever access this variable so the type is irrelevant. Instead, the Scheme portion reflectively accesses the value.

Figure 3.4(b) presents the Scheme implementation of the native methods. Each *native-method* function must accept four arguments in addition to those declared by the Java method. In order, the extra arguments are

1. the current instance of the class – *this*
2. a hash-table mapping symbolic representations of field names to functions that retrieve the values for those fields – *getters*
3. a hash-table mapping field names to functions that modify field values – *setters*
4. and a hash-table providing access to the private methods of the class – *privates*

These arguments allow the native method implementation access to portions of the class structure that would otherwise be inaccessible.

To access the `theImage` field for any `Image` class, the Scheme implementation must first retrieve the accessor function from the hash-table *getters*. The function expects an object of the `Image` class and returns the value of the specified field.

```
class Image {
  private Object theImage;

  public native boolean inside(Image isInside);
  public native Image addLine(Posn start, Posn end, Color c);
}
```

(a) Image.java

```
(module Image-native-methods mzscheme
  (require (lib "image.ss" "htdp") ···)
  ...
  (define (inside-draw2Native.Image-native this getters setters privates image)
    (check
      boolean?
      (image-inside? ((hash-table-get getters 'theImage) this)
                     ((hash-table-get getters 'theImage) image))
      (lambda (v)
        (raise
          (make-java-runtime-exception
            (format "In class Image, inside expected to return a boolean, given ~a" v))))))
  (define (addLine-draw2Native.Posn-draw2Native.Posn-draw2Native.Color-native
            this getters setters privates posn1 posn2 c)
    (new-image
      (add-line ((hash-table-get getters 'theImage) this)
                (Posn-x posn1) (Posn-y posn1)
                (Posn-x posn2) (Posn-y posn2)
                (send (send c toString) get-mzscheme-string))))
  )
```

(b) Image-native-methods.ss

**Figure 3.4**. Image : Native methods

The `inside` method returns a boolean directly from the *image-inside?* function. Internally, Java and Scheme have the same representation for booleans, so no conversion is necessary within the native method. As there are no guarantees that the function will return a boolean, the type of the value must be checked and an error thrown when necessary. The implementation of the *check* function follows.

```
(define (check pred? val thunk)
  (if (pred? val)
      val
      (thunk val)))
```

In the `addLine` method, the `color` object's `toString` method is called. Before passing this value to the *add-line* function, it must be converted. The Java `String` class implementation contains a special method to retrieve a Scheme string from the Java

representation. Also, a new instance of the `Image` is created. While not shown here, the function to create the instance of Image is defined within the Scheme program as well. This function must instantiate the Image object and then call the constructor explicitly.

**3.5.1.2 Java + `dynamic` Implementation.** Figure 3.5 presents the Java + `dynamic` implementation of the `Image` class. The data-structure representing images is stored in the `dynamic` variable `theImage`. Therefore, this value can be used in any manner.

The `import` declaration allows the `Image` to access all of the Scheme image library's functions, including *image-inside?* and *add-line*. Within the Java program, the names `addLine` and `imageInsideP` automatically access the same functions with Java expressible names.

The `inside` method extracts the Scheme object from the parameter and dispatches to the Scheme function. As in the native version, the type of the returned value from *image-inside?* will be checked. In this version, the check is forever tied to the return value of the Java method.

Similarly, the `addLine` method dispatches to the Scheme function, disassembling the arguments to match the Scheme signature. When the `addLine` method accesses the `color` object's `toString` method, the value is automatically converted from a Java `String` into its Scheme counterpart.

**3.5.1.3 Discussion.** For `Image`, the native implementation requires the programmer to take more actions. Interactions between different portions of the same class

```
import scheme.lib.htdp.image

class Image {
  private dynamic theImage;

  public boolean inside( Image isInside ) {
    return image.imageInsideP( theImage, isInside.theImage );
  }

  public Image addLine(Posn start, Posn end, Color c) {
    return new Image(image.addLine(theImage, start.x, start.y,
                                   end.x, end.y, c.toString()));
  }
}
```

**Figure 3.5**. Image: Java + `dynamic`

occur through an indirection required to pass protected information across the language boundary. Values returning to Java should be checked in order to prevent type errors from occurring in odd places within student programs, although there is no enforcement of these checks and they can be forgotten. Values entering Scheme potentially must be converted when data representations do not mesh.

### 3.5.2 View

The View class provides a wrapper over several classes within Scheme, a window frame, editable canvas, drawing element, and an off-screen drawing element for buffering. The class connects all of these Scheme classes and provides a unified view to users. The Scheme canvas class requires an updating callback to direct the behavior when the containing window is resized or obscured.

This section presents the embedding of the Scheme objects into the Java class and the implementation of the `display` method, which instantiates the Scheme objects and opens a blank window.

**3.5.2.1 Native Method Implementation.** Figure 3.6 presents the native method implementation of this class. The Java class contains one `Object` field that will store a Scheme record collecting all four of the objects necessary to interact with the display. Since the Java implementation will never access these values, for the reasons discussed in the `Image` class, collecting the values together simplifies retrieval in the Scheme implementation.

The canvas class expects a function of two arguments for the redrawing callback. This is easily implemented with a local function definition. Accessing and mutating the fields of the `View` class constitutes much of the work required for this class.

This implementation mutates two Java fields, one with type `Object` and one with type `boolean`. The first field is never accessed in Java, so the type does not matter. The second field, `visible`, is used within the Java class and so must be a boolean. While the provided value is a boolean, there is no guarantee that the implementation will not violate Java's type expectation. Since the programmer providing the value is the one who would implement a check around the value, inserting the check provides no more assurance it is the correct type. The same is true for the returned value of the `display` method.

**3.5.2.2 Java + `dynamic` Implementation.** Figure 3.7 presents the Java + `dynamic` implementation. As with the Image class, the View class encapsulates the Scheme objects

```
class View {
  private Object display;
  private boolean visible;
  public native View display( int width, int height);
}
```

(a) View.java

```
(module View-native-methods mzscheme
  (require (lib "mred.ss" "mred") ···)
  (define-struct view (buffer dc canvas frame))
  (define (display-int-int-native this field-accs field-sets privates x y)
    ;Fields and field setters
    (let ((visible ((hash-table-get field-accs 'visible) this))
          (set-visible (hash-table-get field-sets 'visible))
          (name ((hash-table-get field-accs 'name) this))
          (get-display (hash-table-get field-accs 'display))
          (set-display (hash-table-get field-sets 'display)))
      (when visible (send this hide))
    (let* ((buffer (make-object bitmap-dc% (make-object bitmap% x y)))
           (call-back
            (lambda (canvas dc)
              (send dc draw-bitmap
                         (send (view-buffer (get-display this)) get-bitmap) 0 0)))
           (frame (make-object frame%
                                (send name get-mzscheme-string) #f (+ x 10) (+ y 15)))
           (canvas (make-object call-back-canvas% frame null call-back))
           (dc (send canvas get-dc))
           (display (make-view buffer dc canvas frame)))
      (set-display this display)
      (send dc clear)
      (send buffer clear)
      (send frame show #t)
      (set-visible this #t)
      this)))
  )
```

(b) View-native-methods.ss

**Figure 3.6**. View: Native methods

```
import scheme.lib.mred.mred;
import scheme.lib.htdch.graphics.rename;

class View {
  private dynamic frame;
  private dynamic canvas;
  private dynamic dc;
  private dynamic buffer;
  private boolean visible;

//Produces a View with a visible canvas of size x and y
  public View display( int x, int y) {
    if (visible)
     this.hide();

    buffer = rename.newObject(mred.bitmapDcObj,
                    rename.newObject(mred.bitmapObj,x,y));
    buffer.clear();

    class Update {
      public void callBack(dynamic canvas, dynamic dc) {
         dc.drawBitmap(buffer.getBitmap(), 0,0);
      }
    }

    frame = rename.newObject(mred.frameObj, name, false, x+15, y+20);
    canvas = rename.newObject(rename.callBackCanvasObj, frame,
                              rename.emptyList,
                              rename.innerToFunction(2, new Update()));
    dc = canvas.getDc();

    this.clear();
    frame.show(true);
    visible = true;
    return this;
  }
}
```

**Figure 3.7**. View: Java + `dynamic`

as `dynamic` private variables, as they will be accessed individually throughout the class.

The call back for redrawing the canvas class is represented by the inner class, `Update`. However, the Scheme class requires that the supplied callback be a two argument function, which cannot be written within Java without further extensions. The Java programmer resolves this problem by supplying a function, *inner->function*, that wraps an object containing a `callBack` method in a function, which call the method with the correct number of arguments when the function is called. This function is used to wrap all of the callbacks required in the drawing library.

**3.5.2.3 Discussion.** While in this class the lack of ability to define a function in Java suggests some benefits to the native method approach, the native implementation continues to access other portions of the class through an indirect mechanism. Further, there are no checks that values are being set to the correct value and so the programmer relies solely on their own ability to place the correct kind of value. This is not unreasonable, but can be prone to confusing errors. The choice of whether the function or inner class definition better supports interaction can be one of personal preference. However, splitting the important aspects of the implementation across two files can also be a detriment to understanding. This problem could be addressed by extending Java + `dynamic` with the ability to automatically convert objects that implement particular interfaces into appropriate functions.

### 3.5.3 GameWorld

The abstract World class can be written entirely in Java, with all abstract methods and a View `display` object. The GameWorld subclass implements the abstract `animate` method to provide the functionality required of game play. These aspects require that a timer intermittently call an update method to change the representation of the world, and that a callback be registered with the drawing canvas to call an update method to change the representation in response to user input. The MrEd library provides a suitable timer class implementation, which like the canvas class requires a function callback.

In providing a callback to process user input, the teaching library must connect a string from the Scheme program with a Java `String` in the students' programs (this value is the argument to `onKey` in the `World` class). There are no guarantees that the value provided from the Scheme function is indeed a string.

**3.5.3.1 Native Method Implementation.** Figure 3.8 presents the native method GameWorld implementation. As in the previous classes, this version of GameWorld stores

```
public abstract class GameWorld extends World {
  private Object timer;
  private World nextWorld = this;
  public final native boolean animate( int width, int height, int rate );

}
```

(a) GameWorld.java

```
(module GameWorld-native-methods mzscheme
  (require (lib "mred.ss" "mred") ...)
  (define (animate-int-int-int-native this field-accs field-sets privates x y rate)
    (let* ((timer-set (hash-table-get field-sets 'timer))
           (timer-get (hash-table-get field-accs 'timer))
           (nextWorld-get (hash-table-get field-accs 'nextWorld))
           (nextWorld-set (hash-table-get field-sets 'nextWorld))
           (get-display (hash-table-get field-accs 'display))
           (set-display (hash-table-get field-sets 'display))
           (display (get-display this)))
      (let* ((draw-sequence (lambda (o n)
                              (timer-set n (timer-get o))
                              (set-display n (get-display o))
                              (send (get-display this) allowImage #f)
                              (send n draw)
                              (send (get-display this) allowImage #t)))
             (timer-callback (lambda ()
                               (let* ((world (nextWorld-get this))
                                      (new-world (send world onTick)))
                                 (draw-sequence world new-world)
                                 (nextWorld-set this new-world))))
             (key-callback (lambda (key)
                             (unless (string? key)
                               (raise (make-java-runtime-exception
                                        (make-java-string
                                          (format
                                            "Internal error: key must be a string for callback, given ~a"
                                            key)))))
                             (let* ((world (nextWorld-get this))
                                    (new-world (send world onKey-java.lang.String
                                                 (make-java-string key))))
                               (draw-sequence world new-world)
                               (nextWorld-set this new-world))))
             (timer (make-object timer% timer-callback)))
        (send display display x y)
        (send display keyCallBack key-callback)
        (send timer start rate #f) #t)))
)
```

(b) GameWorld-native-methods.ss

**Figure 3.8**. GameWorld : Native methods

the timer object as a private `Object` field and uses the hash-tables to access the various fields and field modifiers.

In the *key-callback* function, the *key* argument must be manually checked and converted into a Java `String`. The `String` library provides the *make-java-string* function to perform this conversion. In the event of an error, the user must explicitly create a Java `RuntimeException` (or suitable subclass) instance to throw. As in creating an instance of an `Image` class, the programmer must create the instance in the two steps of instantiating the class and explicitly calling the constructor. Additionally, due to requirements of interoperating within the DrScheme environment, the exception creation function must set source properties during the creation of the error. The `Throwable` library provides a function to perform some of this translation, but the Scheme programmer must still provide class-specific functionality.

**3.5.3.2 Java + `dynamic` Implementation.** Figure 3.9 presents the implementation of GameWorld for Java + `dynamic` . The `timer` object is a `dynamic` field, and both the timer and user input callbacks use the *inner->function* procedure to translate instances of the classes into functions.

As in the native method version, the argument to `callBack(String)` arises from a Scheme function. When the instance of `KeyCallback` is passed to *inner->function*, the compiler wraps the object to protect its future interactions with Scheme functions.

When the Scheme function invokes this callback method, checks and conversions automatically take place to transform the provided value into a Java `String`. The resulting error, if the value is not a string, will lead back to the GameWorld class with an error in accessing the KeyCallback class.

**3.5.3.3 Discussion.** Each of these classes required more work from the native method implementor than the Java + dynamic. The implementor also had to remember without any assistance all locations where types must be verified and data converted.

Even I, the primary expert in using this native method interface, encountered confusing errors due to supplying incorrect types as an argument to the `onKey` method, and with correctly connecting the Scheme native method library with the portions of the library implemented entirely in Java (such as the Posn class). Such errors did not occur in implementing the dynamic version.

```
import scheme.lib.htdch.graphics.rename;
import scheme.lib.mred.mred;

public abstract class GameWorld extends World {

  private dynamic timer;
  private World nextWorld;

  private void drawSequence( World old, World newW) {
     ((GameWorld) newW).timer = ((GameWorld) old).timer;
     newW.display = old.display;
     display.allowImage(false);
     newW.draw();
     display.allowImage(true);
  }

  public final boolean animate( int width, int height, int rate ) {

   class TimerCallBack {
      public void callBack() {
         World old = nextWorld;
         nextWorld = nextWorld.onTick();
         drawSequence(old, nextWorld);
      }
   }

   class KeyCallBack {
     public void callBack(String key) {
        World old = nextWorld;
        nextWorld = nextWorld.onKey(key);
        drawSequence(old, nextWorld);
     }
   }

    display.display(width, height);

    display.keyCallBack(rename.innerToFunction(1, new KeyCallBack()));
    timer = rename.newObject(mred.timerObj,
                            rename.innerToFunction(0,new TimerCallBack()));
    timer.start(rate, false);
    return true;
  }

}
```

**Figure 3.9**. GameWorld : Java + `dynamic`

# CHAPTER 4

# EXPLORING DYNAMIC

Programs containing `dynamic` variables and references to Scheme programs require more dynamic checks. In particular, compilation must insert the necessary wrappings and runtime checks to validate the type assumptions.

This chapter first presents the locations within a program that require checking `dynamic` variables and protecting Java values. Then the nature of a check is presented, as well as an implementation sketch.

## 4.1 Type Checking `dynamic`

Revisiting the program from Section 3.2.1

```
Fruit getFruit( dynamic tree ) {
  if (tree.hasFruit())
     return tree.pickFruit();
  else
     return tree.getSeeds();
}
```

The compiler analyzes the program and assigns wraps the various uses of `tree` to ensure that the value is an object, with the appropriately named method, and that the method returns a value with the appropriate type. Here, the first method must return a `boolean` and the other two must return instances that match the `Fruit` class.

During the static analysis of a program, the surrounding static type information supplies the dynamic checks that will occur during program execution. This information ensures that when dynamically checked values enter statically verified procedures, the value will cause no primitive runtime failures. If the dynamic value is incompatible with the static expectation, the program will fail at the point that the dynamic value enters the statically verified portion. Runtime errors that were possible in the original program, such as null pointer exceptions, are still possible in the extended program.

The context of a dynamic variable's use provides sufficient information to build type-expectations. Table 4.1 presents the positions in a program that determine which checks are necessary.

In the first five cases, the dynamically checked value is cast (either implicitly or explicitly) to a static type. Since the compiler, after these points, statically checked the validity of the program with respect to the known type, it is necessary that the dynamically-checked value conform to the stated type. The next four cases place a dynamically checked value in a conditional position, which the Java language specifies must contain a boolean. Checking this constraint is simply shifted to a dynamic check. Similarly, in the next three checks, the language requires that the values in these positions be numeric (or Strings in the case of +), so, again, the static check is shifted to a dynamic one.

The remaining four cases present circumstances where dynamic values are used as

**Table 4.1**. Positions to Insert Checks

Program points that specify Checks:

```
KNOWN_TYPE_VARIABLE = dynamic_VALUE;
(KNOWN_TYPE) dynamic_VALUE
KNOWN_TYPE method() { ...  return dynamic_VALUE;}
methodCall(dynamic_VALUE) where method(KNOWN_TYPE)
new KNOWN_TYPE(dynamic_VALUE) where KNOWN_TYPE(KNOWN_TYPE)
if (dynamic_VALUE) ...  ...
(dynamic_VALUE) ? ...  :  ....
while (dynamic_VALUE) ...
for( ...  ; dynamic_VALUE ; ...)
dynamic_VALUE checked-binop primitive_numeric
dynamic_VALUE checked-binop dynamic_VALUE
KNOWN_TYPE_VALUE[dynamic_VALUE]
dynamic_VALUE(..)
dynamic_VALUE.method(...)
dynamic_VALUE.field
dynamic_VALUE[...]
```

Key:

`dynamic_VARIABLE` → Variable with declared type `dynamic`
`dynamic_VALUE` → Value with unknown type
`KNOWN_TYPE_VARIABLE` → Variable with declared type other than dynamic
`KNOWN_TYPE_VALUE` → Value with statically known type
`KNOWN_TYPE` → Non-`dynamic` type
`checked-binop` → One of $+, -, *, /, <, >, <<, >>$, etc. (not $==$)

functions, objects or arrays. In these situations, no specific types are statically known. Instead, a type is invented that must match only the specific use of the value, which is represented with a general mechanism described in Section 4.2. In the case of a method or field access, this expectation reflects that the dynamic value must be an object containing a member of the specified name and the resulting value should be dynamically checked. For method access, the check will also verify that the method can accept the number of arguments presented. In the case that a dynamic value is used as an array, the resulting check will ascertain that the value is an instance of a Java array but places no constraints on the values contained in the array.

In addition to program positions that incur checks, other program positions exist that require adding guards to statically checked values. To preserve type-safety, values with known types require guards when entering positions where the value is dynamically checked. These positions, outlined in Table 4.2, statically hide the known type.

Each case represents an implicit or explicit cast of the known type value to `dynamic` . While the last case may not seem intuitive, the opposite choice of specifying that the dynamic value have the same type as the known value proves too restrictive. This is illustrated with the following example:

```
class One { }
class Two extends One { }
...
One single = new One();
Two double = new Two();
... condition ? single : double ...
```

In this example, it is statically evident that `double` is a subclass of `single`'s class and the resulting type of the expression is `One`. Replacing the type of `single` with `dynamic` changes the type of the expression to `Two`. The value of `single` is not a `Two` so

**Table 4.2**. Positions to Insert Guards

```
Program points that specify Guards:

dynamic_VARIABLE = KNOWN_TYPE_VALUE
(dynamic) KNOWN_TYPE_VALUE
dynamic method() { ...  return KNOWN_TYPE_VALUE;}
methodCall(KNOWN_TYPE_VALUE) where method(dynamic)
new KNOWN_TYPE(KNOWN_TYPE_VALUE) where KNOWN_TYPE(dynamic)
dynamic_VALUE.method(KNOWN_TYPE_VALUE)
BOOLEAN_VALUE ? dynamic_VALUE : KNOWN_TYPE_VALUE
```

this modified program generates a runtime error.

In general, changing a static type to dynamic in a program should not cause a correct program to become incorrect — this would lead to confusion and increase the difficulty of understanding programs containing dynamic. Therefore, the result type of a conditional expression with a dynamically checked value is dynamic. Similarly, an == expression does not constrain the type of a dynamic value as many formerly valid programs would be ruled out through the type check.

## 4.2 Protecting Type-Safety

The type-correctness of flat values, such as numbers, booleans, or characters, can be verified with in-line checks. Higher-order values, such as objects, cannot be so easily verified. In-line checks verify that an object contains the required methods and fields, but cannot verify that a method always returns the correct type when the method may come from a dynamically typed language. To perform higher-order checks (with sufficient information for a precise error report), each object value is wrapped in a contract. A contract in this system follows the ideas of contracts from Findler et al. [23].

### 4.2.1 Contracts

Within a dynamically checked language, such as Scheme, programmers occasionally prefer to enforce type-like constraints on functions and other values. For example, a Scheme library might include the function from Figure 4.1(a). The comment for this function indicates that it expects to receive an integer as the first argument, which will be used to generate a filename for the specified user. If a client calls *save-config* with a value other than an integer, the resulting filename might violate later program expectations or other security concerns (such as with an argument "../../passwords"). To protect against such problems, the Scheme programmer can insert an explicit check of *save-config*'s argument, as seen in Figure 4.1(b).

In the presence of higher-order functions, no immediate check suffices. To illustrate, an additional function in the same library will load a user's configuration once given their id number, see Figure 4.2(a). The implementor of *load-config* not only requires that *get-id* be a function, but also that the function return an integer. A call to (*procedure? get-id*) ensures that *get-id* is a function, but does not ensure that the function will produce an integer. Therefore, *load-config* must additionally check the result of *get-id* before proceeding, as seen in Figure 4.2(b).

```
;; save-config : integer string → void
(define (save-config u-id data)
  (write data (open-output-file (format "config/~a.txt" u-id))))
```

(a) Unchecked Scheme program

```
(define (save-config-checked u-id data)
  (check integer? u-id)
  (save-config u-id data))
;; export save-config-checked to clients as save-config
```

(b) Checked Scheme program

**Figure 4.1**. Checking flat values in Scheme

```
;; load-config : (→ integer) → string
(define (load-config get-id)
  (let ([u-id (get-id)])
    (read (open-input-file (format "config/~a.txt" u-id)))))
```

(a) Higher-order unchecked Scheme function

```
(define (load-config-checked get-id)
  (check procedure? get-id)
  (define (checked-get-id)
    (let ([id (get-id)])
      (check integer? id)
      id))
  (load-config checked-get-id))
;; export load-config-checked to clients as load-config
```

(b) Checking a higher-order value

**Figure 4.2**. Checking functions in Scheme

Implementing these checks for all functions with constraints becomes tedious, and may obscure information that could assist an optimizing compiler. For these reasons, Findler developed a construct similar to an *apply-contract* form for PLT Scheme programmers that annotates values with runtime checks and source information for reporting failures. Figure 4.3 presents constraints for *save-config* and *load-config* using contracts. Specifying these constraints using a new language construct simplifies the creation and modification of these programs, as well as providing more information for compiler optimizations.

### 4.2.2 Contracts and Java + `dynamic`

The Scheme-oriented view of contracts extends naturally to object-oriented programming [24]. As discussed in Chapter 3, requiring the programmer to insert explicit contracts

```
(define save-config-contract
  (integer string . -> . void))
(define save-config-checked
  (apply-contract save-config-contract save-config))

(define load-config-contract
  ((-> integer) . -> . string))
(define load-config-checked
    (apply-contract load-config-contract load-config))
```

**Figure 4.3**. Enforcing constraints with contracts

for fine-grained interoperability adds too much overhead and programmer added contracts may be accidentally omitted. However, the compiler can still use contract technology to insert checks for the programmer.

To demonstrate the operation of contracts, without committing to an implementation strategy, the contracts are depicted as superscripts above the contracted expressions. In Section 3.3.1, a method is written to display dynamically-checked trees written by the students. The following code snippet simplifies the example to show only one contract.

```
void printTree( dynamic tree ) {
  ... tree.age < 100 ...
}
```

The usage of `tree` requires that it have an `age` field containing an integer value. The contract annotated version, including this requirement, adds an abstract *ageField* contract:

$$ageField = \texttt{object}\{\texttt{age} : \texttt{int}\}$$

$void\ printTree(\ dynamic\ tree\ )\ \{$
$\quad \ldots\ tree^{ageField}.age < 100 \ldots$
$\}$

In this notation, *ageField* provides a name for the contract, `object` constructs a contract for an object with the given fields and methods. The fields (and methods) are specified by name, `age`, with the contract following the colon. The contract for a method is `equals` : `object` → `boolean`, method arguments appear to the left of the arrow, and the expected result appears to the right.

This evaluates during program execution, checking the actual value. The steps of this check are shown in Section 4.2.3 using algebraic simplification of expressions, much like earlier models of Java evaluation [25, 26]).

### 4.2.3 Checking Contracts

As in Scheme, the simplest contract applies to flat values. In the following program,

```
int count( int i, dynamic up ) {
  if (up)
    return i + 1;
  else
    return i - 1;
}
```

the `dynamic` variable `up` appears in the condition of an `if`, which requires a boolean value. Assuming the value of `up` at this point is `true`, evaluation first looks up the value associated with the variable, then checks the value against the contract, and finally discards the contract and returns the value (since `true` is a boolean).

$\textbf{if}\ (up^{boolean})\ \ldots.$
$\Rightarrow \textbf{if}\ (\texttt{true}^{boolean})\ \ldots.$
$\Rightarrow \textbf{if}\ (\texttt{true})\ \ldots.$

If, instead, the value associated with `up` is `"true"`, the contract checker detects and reports an error instead of discarding the contract.

$\textbf{if}\ (up^{boolean})\ \ldots.$
$\Rightarrow \textbf{if}\ (\texttt{"true"}^{boolean})\ \ldots.$
$\Rightarrow error$

In this scenario, the blame for the contract violation lies with the client of `count`, who by using `count` accepted responsibility for providing the method with a boolean instead of a string.

For higher-order contracts that cannot be checked immediately, the checks must occur in a coherent order. In Figure 3.5, the method `inside` calls the `dynamic` function `imageInsideP` (imported from `scheme.image`) where the context requires that the function return a boolean. The contract for this use is

$$imageInsideP^{(\texttt{any any}\rightarrow\texttt{boolean})}(\ theImage,\ isInside.theImage)$$

This contract states that the `imageInsideP` function must accept two arguments of any type and return a boolean. The *any* contract is satisfied by all values and corresponds to values with type `dynamic` and no contextual restrictions. When checked, this contract verifies that the `imageInsideP` variable refers to a function value with the correct arity. If this check succeeds, argument and result contracts distribute to the actual arguments of the function and to the result of the application.

$imageInsideP^{(\texttt{any any}\rightarrow\texttt{boolean})}(\ theImage,\ isInside.theImage)$
$\Rightarrow (imageInsideP(\texttt{theImage}^{\texttt{any}},\ \texttt{isInside.theImage}^{\texttt{any}}))^{\texttt{boolean}}$
$\Rightarrow \ldots$

Resolution of the contracts on the arguments proceeds as in the `up` example, as will the resolution for the result once the function application results in a value.

In the event that any portion of the above checks fails, assigning blame is more complicated than the *boolean* contract described previously. If `imageInsideP` is not a function with the correct arity, then the supplier of this value is to blame, in this case the Scheme *image* library. As the client of `imageInsideP`, the Java `Image` class is to blame for any defects in the arguments presented to the function. And, once again, the supplier of `imageInsideP` holds responsibility for the result of matching the contract, as the supplier of the function is also the supplier of the resulting value.

In this interaction, the two parties involved not only represent different libraries, but also different languages. The position of the contract represents the point at which values flow between Java and Scheme. In general, the language in control of the evaluation at the point where values pass between languages is responsible for those values. Since function arguments flow in the opposite direction from function results, the blamed party for a function argument is the opposite of the blamed party for a function result.

In general, the languages interacting at dynamically-checked points can be Java + dynamic and any arbitrary language (including Java + dynamic itself). For simplicity of presentation, I restrict blame assignment to Scheme and Java (+ dynamic). Additionally, individual libraries and classes will not be identified. Contracts that blame Java are annotated with *JS:*, while contracts blaming Scheme gain the annotation *SJ:*. To explain further, the previous example is revisited with the appropriate annotations.

$imageInsideP^{SJ:(\texttt{any any}\rightarrow\texttt{boolean})}(\ theImage,\ isInside.theImage)$
$\Rightarrow (imageInsideP(\texttt{theImage}^{JS:\texttt{any}},\ \texttt{isInside.theImage}^{JS:\texttt{any}}))^{SJ:\texttt{boolean}}$
$\Rightarrow \ldots$

During the second step of the modified example, the annotation moves onto the arguments, with the polarity reversed to indicate a change in responsibility, and result of the function call. If `imageInsideP` is not a function, an error occurs during this step blaming $S$. The argument and result contracts bear no further relevance.

Like procedure contracts, object contracts must also distribute the method argument and result contracts for the specified method during method invocation. When, in Fig-

ure 3.9, an instance of the `KeyCallBack` class is passed into the dynamic `innerToFunction` function, the value gains an object contract. For this call, the relevant contracts appear below:

$$keycallback = \texttt{object}\{\ callBack : \texttt{String} \rightarrow \texttt{void}\}$$

$$innerToFunction^{SJ:(\texttt{int}\ keycallback \rightarrow \texttt{any})}(1, \textbf{new}\ KeyCallBack())$$

Evaluating the call to `innerToFunction` instantiates the `KeyCallBack`, checks that `innerToFunction` is indeed a function of two arguments, and then distributes the argument and result contracts before calling the function. This means that the `KeyCallBack` instance flows into Scheme annotated with the contract $JS : keycallback$. This annotation remains with the value within the resulting **lambda** expression, and the contract is checked on any access to the object.

The `Canvas` class calls the function returned from `innerToFunction`, which in turn calls the `callBack` method of the `KeyCallBack` class using the Scheme **send** form (which performs a method call). At this point, the contract wrapping the specific instance (hereafter to be referred to as `keyobj`) comes into play.

(**send** $\texttt{keyobj}^{JS:keycallback}$ "left")
$\Rightarrow$ (**send** `keyobj` *call-back* $\text{"left"}^{SJ:String})^{JS:void}$
$\Rightarrow \ldots$

At the call site for the method `callBack`, the contract on the method moves the contract for the argument to the appropriate value, and places the contract for the result on the **send** expression. In this circumstance, the Scheme string satisfies the contract entering Java, a conversion is performed, and void is returned from Java. If the value passed into `callBack` were not a String, the Scheme program would bear the blame of the misuse. The resulting error message pinpoints the location where the Java value entered Scheme, that the violation occurs within the access to the `KeyCallBack` class's `callback` method, within the specific Scheme library accessing the value.

In a circumstance where the value returned to Scheme from a Java method is another instance of a Java object, the value is wrapped with an appropriate context. For example, assuming that the callBack method returns a `World` object, instead of nothing, the *keycallback* contract changes to:

$$keycallback = \texttt{object}\ \{\ callBack : \texttt{String} \rightarrow world\ \}$$

The result of calling the method is an object wrapped with a *world* contract, thus any value entering Scheme from Java is wrapped with a contract to enforce the type requirements.

As mentioned in Section 4.2.1, Scheme libraries may provide checked versions of their values by attaching contracts. Each contract may be unrelated to the contract attached to the value by Java. The same basic technique applies to the contract checking, where both contracts are checked at the requisite point. Since the two contracts originated in different locations, there are now three potential sources of an error: Java does not meet Scheme's contract, Scheme does not meet Java's contract, or the two contracts do not match. In the third case, neither the Java program nor the Scheme program is expressly to blame. Instead, a third party, named $C$, is involved in the program to mediate the composition of the two languages. This party is blamed for any mismatch between the Java and Scheme contracts.

In the Scheme library *rename* (used in the Java libraries presented in Chapter 3), there is a function *inner->function* that wraps method calls within a lambda. A reasonable contract for this function checks that the argument is an object containing the *call-back* method, with an unspecified number of arguments and promises to return a function. In the syntax presented thus far, this contract is

$$\mathit{callbackobj} = \texttt{object}\ \{\ \mathit{call\text{-}back} : \texttt{any*} \rightarrow \texttt{void}\ \}$$

$$\mathit{inner\text{-}{>}function\text{-}cont} = \mathit{callbackobj} \rightarrow (\texttt{any*} \rightarrow \texttt{void})$$

For a simple example of a disagreement between a Java and Scheme contract, the following program generates a runtime error:

```
1 + innerToFunction(new KeyCallBack())
```

The type requirements of the Java `+` expression dictates that the left hand argument (the result of calling `innerToFunction`) be an integer. Therefore, the Java contract is *keycallback* $\rightarrow$ `int`; however, the Scheme program provides the *inner->function-cont* contract, specifying that the returned value will be `void`.

To enforce both contracts on the same value, they each appear in the exponent on `innerToFunction`, with $C$ as the opposite party:

$$\texttt{1} + \mathit{innerToFunction}^{SC:(\mathit{callbackobj} \rightarrow \mathit{void}), CJ:(\mathit{keycallback} \rightarrow \mathit{int})}(\textbf{new}\ \mathit{KeyCallBack}())$$

Since the value initially flows from Scheme to Java, the $SC$ annotation appears on the Scheme contract, indicating that Scheme is initially responsible for the value itself, and

the $CJ$ annotation appears on the Java contract, indicating that Java is responsible for arguments flowing into the value.

Following the same simplification steps as before, but with the new contracts, gives

$$\Rightarrow 1 + \mathit{innerToFunction}^{SC:(\mathit{callbackobj} \to \mathit{void}),CJ:(\mathit{keycallback} \to \mathit{int})}(\texttt{objref})$$
$$\Rightarrow 1 + \mathit{innerToFunction}^{(CS:\mathit{callbackobj} \to SC:\mathit{void}),(JC:\mathit{keycallback} \to CJ:\mathit{void})}(\texttt{objref})$$
$$\Rightarrow 1 + (\mathit{innerToFunction}(\texttt{objref}^{JC:\mathit{callbackobj},CS:\mathit{keycallback}}))^{SC:\mathit{void},CJ:\mathit{int}}$$

At this point, the two contracts on `objref` are compatible, so the composer can never be blamed. The contract on the results, however, are different. Assuming that *innerToFunction* matches its Scheme contract, eventually evaluation produces the following (where *void* is a special value in Scheme that has no operations, unlike `void` in Java):

$$\Rightarrow 1 + \mathit{void}^{SC:\mathit{void},CJ:\mathit{int}}$$

Now, since the value is indeed void, the Scheme contract can be discarded, leaving

$$\Rightarrow 1 + \mathit{void}^{CJ:\mathit{int}}$$

which aborts the program, since void is not an integer. In this case, $C$ is blamed, indicating that a mis-match between the two contracts was detected.

In a real implementation, the annotations $S$, $J$, and $C$ point to the program points where the code first began interoperating, so that the contract failure pinpoints a specific expression, and not merely a specific side of the interoperating code.

## 4.3 Adding Support for `dynamic`

While many implementation techniques exist for dynamic, this section presents a technique using mirrors to support the checks and guards across the language boundaries.

### 4.3.1 Mirrors and Reflection

Support for reflection is usually implemented with a combination of virtual-machine support and a reflection API, where reflection support is built into every declared class. A mirror-based implementation of reflection, in contrast, supports reflection operations separately from each declared class.

One way to implement mirror-based reflection is to generate a mirror class for each declared class. An instance of the mirror class embeds the base objects, and it contains all information that is needed to support reflection.

For example, the drawing library contains a class, `Posn`, to represent coordinates.

```
class Posn {
  Integer x,y;
}
```

In a Java-style reflective system, the `Posn` class automatically includes methods to support reflection operations. With mirror-based reflection, the `Posn` class contains no such methods, and reflection operations are instead supplied by a class like `PosnMirror`:

```
class PosnMirror extends ClassMirror {
  Posn p;
  Object getField( String name ) {
    if (name.equals("x"))
      return p.x;
    ...
  }
}
```

A mirror encapsulates all reflection operations, it is separate from the base object, and its structure reliably reflects the structure of the core object [27].

### 4.3.2 Implementing `dynamic` with Mirrors

Mirrors provide the reflective framework to support dynamically-checked accesses of Java objects. Returning to the `Image` class of the drawing library,

```
class Image {
  dynamic theImage;

  Image( dynamic init ) {
    theImage = init;
  }

}
```

the converted signature for `Image`'s constructor using a `Mirror` is

```
Image(Mirror init)
```

and the type of the corresponding field becomes `Mirror`. In an initialization of the `Image` class with a non-`dynamic` argument, such as

```
new Image( new Bitmap("") )
```

the compiler must insert a coercion to the argument's mirror.

```
new Image(new BitmapMirror(new Bitmap("")))
```

Given a `Bitmap` declaration

```
class Bitmap {
  ....
  boolean inside(Bitmap img) { .... }
}
```

the compiler generates a mirror class.

```
class BitmapMirror implements Mirror {
  final Bitmap orig;
  BitmapMirror(Bitmap _orig) { orig = _orig; }
  ...
}
```

The main operation that the compiler needs on the `Mirror` interface is `call`, which dynamically locates a method by name and argument count, and then applies an array of `Mirror` values to produce a `Mirror` result. Thus, the implementation of `BitmapMirror` includes a `call` method as follows.

```
class BitmapMirror implements Mirror {
  ...
  Mirror call(String name, Mirror[] args) {
    if (name.equals("inside") && (args.length==1)) {
      ....
    } else
      raise new Error("method not understood");
  }
}
```

To implement the dynamic call, the `call` method of the class `BitmapMirror` must unpack its arguments (raising an error if an argument does not have a suitable type), call the original method, and then pack the result as a `Mirror`.

```
Mirror call(String name, Mirror[] args) {
  if (name.equals("inside") && (args.length==1)) {
    Bitmap img;
    if (args[0] instanceOf BitmapMirror)
      img = ((BitmapMirror)args[0]).val();
    else
      raise new Error("bad argument");
    Boolean result = orig.inside(img);
    return new BooleanMirror(result);
   } else
    raise new Error("method not understood");
}
```

This `Mirror` implementation includes dynamic checks that are just like contract checks. The check that the argument is a `Bitmap` is like the check in *callbackobj*. The coercion

of `result` to a mirror is much like wrapping *get-id* with *check-get-id*; it can be viewed as applying the contract "must be used as a boolean" to the result of the method.

A method call on a `dynamic` value translates to a dispatch to the `Mirror`'s `call` method. So within the image class, a call to `inside` translates to:

```
theImage.call("inside", new Mirror[] {img.theImage} )
```

Field access will be handled similarly.

### 4.3.3 Static Types from Dynamic Values

In the Scheme examples, only definitions acquire contracts. In Java with `dynamic`, variable declarations and dynamic expressions both acquire contracts. For example, in the `GameWorld` class, the call `timer.start(rate, false)` implies a contract on the `timer` value:

$$startContract = \texttt{object}\ \{\ start : \texttt{int boolean} \rightarrow \texttt{any}\ \}$$

Taking representation issues into account, the call to `timer`'s method must also coerce arguments to `Mirror`s and coerce the result from `Mirror` (in the cases where the result is also used). Just as for declaration-side contracts and coercions, user-side contracts and coercions can be naturally packaged together in a compiler-generated `Unmirror` class, as follows.

```
....
  Mirror timerOrig;

  ....
  new StartUnmirror( timerOrig ).start(rate, false);
....
final class StartUnmirror {
  Mirror m;
  StartUnmirror (Mirror _m) { m = _m; }

  void start(int rate, boolean rerun) {
    Mirror[] args = new Mirror[2];
    args[0] = new IntegerMirror(rate);
    args[1] = new BooleanMirror(rerun);

    /* Might raise a dynamic exception */
    Mirror result = m.call("start", args);
  }
}
```

As in `BitmapMirror`, the checks and coercions in the class `StartUnmirror` are based on the static types surrounding the `timer.start(rate, false)` call. Specifically, the first

argument is packaged using `IntegerMirror` because the type of the argument `rate` is `int`, and similarly for the second argument. If the result of the function were used, a check of the `Mirror` result value would be necessary.

When a `dynamic` value is stored as an instance of a specific class, for instance

```
dynamic buffer;
....
JBuffer b = buffer;
```

the `dynamic` value is wrapped with an unmirror specific to the Java class. So, in the above example, the `buffer` object (an instance of the `Mirror` interface), will be wrapped in an instance of UnmirrorJBuffer that contains all of the methods of the `JBuffer` class, and appropriately mirrors and packages the arguments to pass to the wrapped `Mirror` instance.

The generated `StartUnmirror` class above has a `final` declaration to emphasize that it is completely under the control of the compiler, which might choose to eliminate the instance and inline the method call. An inlining optimization produces checks immediately around the dynamic `start` call, which is what a programmer would intuitively expect. In contrast, the `IntegerMirror` object cannot be eliminated in general; it encapsulates obligations for a dynamic implementation of `start`, to ensure that `start` uses the integer safely. This wrapper is consistent with the intuition that statically typed objects passed to dynamic code must be wrapped with checks to guard the object.

## 4.4 Related Work

### 4.4.1 Mixing Dynamic and Static Types

Strongtalk [28] adds an optional static type system to Smalltalk [29]. On the boundary between typed and untyped expressions the compiler either assumes a type or relies on an annotation from the programmer. Unlike Java + `dynamic` , these type assumptions are not validated at runtime, so that if a type annotation or assumption is incorrect, typed operations can generate dynamic type errors.

The Amber programming language [30] also mixes static and dynamic type checking. Values with statically checked types can be placed into `Dynamic` wrappers, in which the static type information is disregarded. During program execution, interaction with these values is checked to conform to the static type knowledge. When a value without a statically known type was placed within a `Dynamic` construct, the programmer must explicitly inform the system of the type expectations of the value. Unlike Java + `dynamic` ,

interaction with Dynamic values is not identical with interaction with statically checked values and required programmer intervention in cases where the value never had a static type (such as when a value enters from Scheme within, or when a value enters through loading in Amber).

A proposed extension to ML allows for a `Dynamic` type constructor [31, 32], which is similar to the constructor within Amber as well as the `dynamic` declaration. The `Dynamic` constructor encapsulates data with unknown types, with the intent that untyped data arise from I/O calls, as opposed to other programming languages. In order to extract data from a `Dynamic` value, programmers first explicitly test the type of the value. Only data with named types can be extracted. As in my system, these types are checked at run time, but a programmer must explicitly cast the data and provide datatype definitions for all values. Unlike for Amber and Java + `dynamic` , a value with a statically known type is not protected when entering a `Dynamic` constructor.

Work on embedding languages by Benton [33] and Ramsey [34] provides connections between statically typed languages and embedded dynamically typed languages. For both systems, when a value from the dynamically typed language is passed into the statically typed language, the system performs an immediate check of the value. The expected type is derived from a specification either written by the programmer or provided by the system library. This system requires more effort on the part of the programmer to specify the types of the dynamically typed values as well as tying the dynamically typed program to the static type system of the other language.

A proposed extension to the Java Virtual Machine [35], `invokedynamic`, would add the ability to dynamically invoke methods without the static knowledge that such a method exists [36]. This instruction would permit an extension to Java similar to Java + `dynamic` , but would not fully eliminate the need for mirrors or unmirrors within the resulting system to ensure the proper treatment of values with statically known types entering dynamically checked variables.

### 4.4.2 Language Interoperability

The Portable Common Runtime [37] provides a shared address space for programs written in multiple languages, along with support for channels that different programs can use to communicate. Similar approaches, including the Mercury system [38] and HRPC [39], supplied connections through remote procedure calls and IO techniques built-in to the support libraries. Java + `dynamic` also uses a shared memory system for the

languages, so that low level memory management is not required of the programmer. These earlier systems did not provide data checks and conversions on language boundaries, nor were interoperability calls fine-grained.

Component models, such as COM [40], CORBA [41], SOM [42], and DOM [43], support language interoperability through a shared interface definition language (IDL). Each IDL is effectively a universal interface for data interchange, so that more than two languages can communicate with an application, but this “least common denominator” approach limits the granularity of interoperability. Furthermore, the component model is generally specified outside the programming language, so that it is relatively opaque to compilers and analysis tools.

The Microsoft .NET framework [44, 45, 46] is similar to component models, but interoperating languages are all compiled to a Common Intermediate Language (CIL). This approach makes interoperability more apparent to the compiler, but also limits implementation strategies for languages to those that fit well into CIL.

SWIG [47] bridges C-level libraries and higher-level (often dynamic) languages by automatically generating glue code from C and C++ sources. Programmers can implement new modules for SWIG to support bridges from C to new programming languages, but only to the degree supported by a foreign-function interface in the non-C language. In short, SWIG supports interoperability much like component models or .NET, but with C as the universal API.

The nlffi-gen [48] foreign-function interface for ML produces ML bindings for C header files. The system also provides an encoding of C datatypes into ML so that ML programmers may directly access C datatypes. Marshaling of data occurs within the ML program, written by the programmer, where necessary. The system does not provide support for ML data representation into the C code. Interoperating between ML and C is easier for the programmer, although type-safety and data representation are still a concern for the programmer.

Furr and Foster have developed a system that analyzes C foreign-function libraries for OCaml [49]. Their tool uses OCaml type information to statically locate misuses of the OCaml data within C. In a sense, this tool effectively bridges the gap between a strongly typed language and a weakly typed language by strengthening the latter’s type system.

# CHAPTER 5

# BUILDING THE SYSTEM

To provide customized error messages throughout the development process, the compiler for the ProfessorJ languages (including Java + dynamic) begins with Java source. It then proceeds with a standard type-checking pass, and finally translates the Java program into an equivalent PLT Scheme program that includes source-location information for the original program. Maintaining the source location allows run-time errors to report the position of the error, and also allows greater interaction with DrScheme tools.

Supporting multiple languages with the same implementation requires that the front-end of the compiler, including the parsing and type-checking phases, support different rules, depending on the language of the current program. Further, the compiler must be able to switch the language for different portions of the program, as imported classes may be written in a different Java language. These requirements drive different consideration within the implementation than a compiler for only full Java.

The addition of `dynamic` into the language also requires differences in the type-checking phase over nonparametric Java type-checking. While a value with type `dynamic` cannot cause a type-checking error, the type-checker is responsible for determining the expected types of dynamic uses and the locations where values with known types become dynamically checked.

When translated into Scheme, the Java program contains Scheme functions, constants, and classes. While standard Scheme does not contain a mechanism for class definition, the PLT Scheme variant supports a class system implemented with macros. This extension allows greater interoperability between Java and Scheme, as well as shifting the work of implementing a language construct away from the task of compiling Java so that the two concerns can be separated. Similarly, ProfessorJ uses the Scheme contract macros to implement dynamic checks.

## 5.1 Compiling for Multiple Languages

The back-end of the ProfessorJ compiler is language-agnostic, with no knowledge of which Java subset is being compiled. The front-end cannot be language-agnostic, and must support the differences between the languages through parsing and type-checking technologies.

### 5.1.1 Parsing

The ProfessorJ compiler supports five syntactically different, although similar, languages. Despite the similarities, the parsers cannot all reside in one Yacc style parser specification. Additionally a Yacc parser is not an ideal technology for including clear error messages. So, the system contains four Yacc-style parsers, three of which support the pedagogic languages and one which supports both the Java language and Java + `dynamic` and one special parser for producing error messages.

The parser for the full language, which contains a production for the `dynamic` keyword when an appropriate flag is set, comes from the Java specification [50] first edition, augmented with productions from the second edition. Each of the pedagogic parsers follows from this specification, with productions removed or simplified as necessary to only allow the syntactic constructs of the current language.

While each parser does contain a default error message in the event that the program does not meet the language specification, this default message should never appear for the pedagogic languages. The Yacc parsers build the abstract syntax tree, but when a program does not properly parse, a specialized error parser is called. The error parser is a hand-written, top-down, recursive-descent parser that supports all three pedagogic languages and encodes expert knowledge of student errors within the program logic.[1]

The error parser disables productions based on the language being compiled, which permits the one parser to support all three languages. It is also connected to a primitive keyword spell checker to distinguish misspellings from misplaced identifiers. Some productions identify common programming errors instead of the correct syntax to provide specific error messages. Where possible, error messages provide guidance regarding the correct form of the program, such as reminding users of the proper structure of a field. The knowledge encoded within the parser, including common error situations, advice as

---

[1] Both the full language and Java + dynamic rely on the Yacc parser's default message, and therefore are not intended for student use at this time.

to the correct syntax, and error message vocabulary, is tightly connected to the three subsets that the compiler presently supports.

While this parsing solution supports the current system with three languages, it is not ideal for building a parser from a declarative specification.

### 5.1.2 Type Checking Multiple Languages

The type-checker operates in two phases. The first phase checks the structure of classes, while storing signature information for each class for later use. The second phase checks the initializations of fields and the bodies of all methods, using the stored class signature information where necessary. Both phases use function parameters to keep track of language restrictions to apply, which allows different portions of the program to be checked in different languages.

Checks for many language constructs can be wholly language-agnostic, as the different parsers preclude their presence in languages that do not support them. Such features include checks of arrays, loops, casts, and others. Other checks are required for all languages, such as only implementing interfaces, not instantiating interfaces, and not attempting to subtract a number from an object. Finally, some property checks should only occur within specific language levels.

In language-specific cases, checks are placed behind a guard that only executes the check for the correct language level. For example, the check that all thrown exceptions are either caught or declared occurs only in the full and dynamic languages, while the check that fields and methods do not share the same names occurs only in the pedagogic language levels.

Static analysis can take advantage of the language level knowledge to suggest optimizations to the final pass of the compiler with less actual analysis. For example, in ProfessorJ Beginner, the compiler automatically suggests that all methods be optimized for tail-calls, since all `returns` within these methods must be in tail position.

### 5.1.3 Error Messages

Like the language-specific static checks, the vocabulary of error messages is modified by the current language level. For example, error messages in the Beginner and Intermediate languages refer to creating a new instance of a class, while in Advanced the same error may refer to creating a new instance of an array. Some error messages present entirely different content depending on the language level. For example, in the Beginner

language, putting a nonstatement in a statement position generates an error message listing all (two) of the possible statements in Beginner. The same error situation cannot have the same error message in the other languages as there are too many statements to reasonably list.

To provide the most specific error possible, extra information is sometimes retained. In the following cases, extra information provides clearer error messages:

- In a constructor, if another constructor is to be invoked, the invocation must be the first statement in the constructor. While checking the remainder of the constructor body, it is unnecessary to know whether the super call has already been called. However, this information can clarify the resulting error message (to say both that the constructor call must be first, and that it cannot occur multiple times).
- When a field access fails for an array, knowing that the accessed object was an array can permit the error message to explicitly state that arrays have only the length field.
- Further, knowing that an expression should return an integer because it is in an array access position can allow the error message to specifically state the reason why the expression must be an integer.
- Storing the name of the current class, or the parent class or interface where an inherited method arises from, can allow many error messages to pinpoint more information in a program which compiles multiple classes in the same file.

Other information can also be stored for providing clearer error messages, requiring only that the error messages receive sufficient focus during development.

## 5.2 Type Checking `dynamic`

Section 4.1 presented the locations that constrained the permitted types of dynamically checked values. Implementing Java + `dynamic` requires that the static type checker store the expected type of a particular use of a dynamic value. Then, the next phase of the compiler uses this information to generate code that contains the required check.

Gathering this information requires adding checks into the process of checking the constructs listed in Tables 4.1 and 4.2. The checks determine when an expression with type `dynamic` occurs within the production, assigning the expected type of the expression or recording the known type of expressions when a guard is necessary.

Adding function application of `dynamic` requires a more involved modification. Prior to this addition, a flat method call such as `draw()` triggered a search in the current class

for a `draw` method. After the change, this search must first look for a `draw` parameter with type `dynamic`, then either a method or field named `draw`. The choice is to first search methods and then fields for `draw`. With this ordering, programs cannot use a `dynamic` field as a function when the class contains a method with the same name.

Expressions of the form `a + b`, where both `a` and `b` have been declared `dynamic` , cause a small dilemma. The `+` operator could allow the arguments to be any of Strings, characters, numbers, etc. The type ought to dictate which form of `+` is intended, but these are not known until runtime. Three possible solutions exist: defer to the expected type of the expression; assume that the values are numbers unless otherwise indicated (or Strings); defer all decisions in this circumstance until runtime. The first choice is only an option if the expected type is not also `dynamic`. The current implementation chooses to defer to the expected type when one is available, and assume numbers when no type is available.

Another related problem arises in selecting overloaded methods. When the compiler can reduce the list of possible methods to one by arity, the unique choice is used. When there are multiple possible methods with the correct arity, if any of the methods expects a dynamic parameter in the position of the dynamic value, this method is chosen. In all other circumstances, the compiler indicates that there is not enough information to select the method. Programmers can remedy this with a cast, as is already necessary in other circumstances when resolving overloading. While a runtime decision may provide enough information, in the circumstances where a cast is necessary in a static world, the program would fail at runtime to be able to choose between the methods. This situation is also avoided by requiring the programmer to resolve the overloading.

## 5.3 Compiling to Scheme

For the sake of easy interoperability from Scheme into Java programs, it is important the the compiled Java support similar interfaces as Scheme programs, including the import mechanism, object access, and class inheritance. This motivation directed several choices in the output of a Java class.

### 5.3.1 Compilation Units

A single Java source file typically contains one `public` class or interface. Often, the file itself corresponds to a compilation unit, so that one `.java` file can be compiled to one `.class` (or, in this compiler, to one `.scm` file).

In general, however, reference cycles can occur among `.java` files, as long as they do not lead to inheritance cycles. Thus, the compilation unit actually corresponds to several mutually dependent `.java` files. For example, one class may refer to a field of another class, and compiling this reference requires information about the structure of the referenced class. In contrast, merely using a class as an identifier's type does not necessarily require information about the class, especially if the identifier is not used.

More concretely, the code in Figure 5.1 corresponds to three source files, one for each class. Compiling `Empty` requires knowledge of the superclass `List`, while compiling `List` requires knowledge of `Empty` for the constructor call. Similarly, `List` refers to `Cons` and `Cons` refers to `List`. Thus the three classes must all be compiled at the same time. This style of cyclic reference appears frequently in real Java code.

Java's packages are orthogonal to compilation units because a group of mutually dependent `.java` files might span several Java `package`s. Furthermore, a mutually dependent group of files rarely includes all files for a package, so forcing a compilation unit to be larger than a package would lead to needlessly large compilation units. Finally, in most settings, a Java package can be extended by arbitrary files that simply declare membership in the package, which would cause an entire package to recompile unnecessarily.

To a first approximation, the ProfessorJ compiler produces a single Scheme **module** for each collection of mutually dependent Java sources, where **module** is the unit of compilation for PLT Scheme code [51]. Each class used by, but not a member of, the dependent group is **require**-ed into the **module**. The Java specification [50] requires that each class be initialized and available prior to its first use, which the **require** statement ensures.

The **module** is also a unit of organization at the Scheme level, where Scheme programmers **require** modules as Java programmers `import` classes. Allowing Scheme programmers to access individual Java classes as modules aids interoperability from the Scheme perspective.

Thus, the compiler actually produces $N + 1$ modules for $N$ mutually dependent Java sources: one that combines the Java code into a compilation unit, and then one for each source file to re-export the parts of the compilation unit that are specific to the source.[2] Thus Scheme and Java programmer alike import each class individually. For example,

[2] If a class is not a member of any dependency cycle, then the compiler produces only one module.

```
public abstract class List {
  public abstract int length();

  public static void main() {
    Test.test(new Empty().length(), 0);
    Test.test(new Cons(1,
                       new Empty()).length(),
              1);
  }
}

public class Empty extends List {
  public int length() { return 0; }
}

public class Cons extends List {
  int car;
  List cdr;
  public Cons( int c, List cdr ) {
    this.car = c;
    this.cdr = cdr;
  }
  public int length() { return 1 + cdr.length(); }
}
```

**Figure 5.1**. Java program with dependency cycle

compiling Figure 5.1 results in four modules: a composite module that contains the code of all three classes and exports all definitions, a `List` module that re-exports `List` and `main`, an `Empty` module that re-exports `Empty`, and a `Cons` module that re-exports `Cons` and field-relevant information.

In practice, most groups of mutually dependent files are small, so that the resulting compilation units are manageable. This is no coincidence, since any Java compiler would have to deal with the group as a whole. In other words, this notion of compilation unit is not really specific to this compiler. Rather, having an explicit notion of a compilation unit in the target language has forced a precise understanding of what compilation units are in Java, and to reflect those units in the compiler's result.

The compiler produces an additional file when generating Scheme from Java code. This extra file contains class signature information, such as the types and names of fields and methods, the implemented interfaces, and the parent class. This information is used to process other compilation units that depend on the Java class. Other Java compilers use the `.class` files to retrieve this information; however, this information does not fit

as naturally into the Scheme modules. Future versions of the compiler will likely embed the type information into the **module** in much the same way that compile-time macros are stored in **module**s.

### 5.3.2 Classes

A Java class can contain fields, methods, nested classes (and interfaces), and additional code segments, each of which can be static. PLT Scheme classes are similar to Java classes, except that they do not support static members. Nevertheless, a static member closely corresponds to a Scheme function, value, or expression within a restricted namespace, i.e., a **module**, so static Java members are compiled to these scheme forms.

The Scheme classes contain overridable methods and nonshadowable fields, both of which can be either public or private. They implement Scheme interfaces (collections of method names) and extend Scheme classes, with `object%` serving as the inheritance base. Expressions placed in the top-level of a Scheme class are evaluated when the class is instantiated. An instance of a class is created with a **new** form that provides **init** variables with their values.

The top-level expressions serve the same purpose as a single Java constructor, and no explicit constructor form is provided. However, a Java class can contain multiple constructors, preventing a direct translation from a Java constructor to a sequence of top-level expressions. Instead, constructors translate into normal methods within the Scheme class. A Java `new` expression translates into an invocation of the Scheme **new**, followed by a call to the constructor method.

Overall, the initialization sequence of the compiled class

1. provides fields with default values based on their types;
2. mutates the field to the initialized value, where applicable;
3. executes any top-level block statements;
4. calls the constructor method specified by the argument types.

This behavior adheres to the guidelines for class instantiation provided by Java's specification [50].

Scheme additionally does not support `abstract` classes. These classes translate into full Scheme classes with stub methods for any `abstract` methods.

Figure 5.2 presents a sketch of compiling two Java classes into Scheme. If either class implemented an interface, the name of that interface would appear within the parens to the right of the parent class.

### 5.3.3 Fields and Methods

The member fields of a Java class translate into **field** declarations. Scheme fields must either be initialized at their definition site or through initialization arguments passed to the **new** form. A `static` Java field, meanwhile, translates into a Scheme top-level definition. Thus, the fields in the `Animal` class

```
static int total;
int myId;
```

become, roughly

---

```
abstract class Animal {
  static int total;
  int myId;
}
class Fish extends Animal {
  Fish( ) { ... }
}
... new Fish( ) ...
```

(a) Java classes

```
(define Animal
  (class Object ()
     ...
     ))
(define Fish
  (class Animal ()
      (define (Fish~constructor) ...)
      ...
      ))
 ... (let ((new-obj (new Fish)))
        (send new-obj Fish~constructor)
        new-obj) ...
```

(b) Scheme compilation of Java classes

**Figure 5.2**. Compiling Java classes

---

(**define** *total* 0)

and

(**define** *Animal*
  (**class** *Object*
    (**field** (*myId* 0)) · · ·))

However, the above translation does not connect the variable `total` to the containing class `Animal`. If multiple classes within a compilation unit contain a static field `total`, the definitions would conflict. For nonstatic fields, Scheme classes do not allow subclasses to shadow field names again potentially allowing conflicts. Unlike Java, Scheme classes do not provide distinct namespaces for fields and methods, so to avoid conflicts the compiler appends a *˜f* to the field names. Therefore, the `total` name is combined with the class name and *˜f*, forming the result as *Animal-total˜f*, and `myId` becomes *Animal-myId˜f*. Note that Scheme programmers using these names effectively indicate the field's class.

To access Scheme fields from outside the class, it is necessary to use special functions built using the class. The compiler uses these to generate a mutator and accessor function for `myId`; additionally, it generates a mutator function for `total`. Since the **module** form prohibits mutating an imported identifier, the mutator function *Animal-total-set!* provides the only means of modifying the static field's value. If the static field were `final`, this mutator is not exported. Also, instance field mutators are not generated when they are `final`. Thus, even without compile-time checking, Scheme programmers cannot violate Java's `final` semantics.

Similarly, instance methods translate into Scheme methods and static methods into function definitions with the class name appended, but the name must be further mangled to support overloading. For example, an expansion of the class `Animal` in Figure 5.3(a) contains two methods named `feed`, one with zero arguments, the other expecting one integer.

As seen in Figure 5.3(b), the method `feed(int)` translates into *feed-int*, and `feed` translates into *feed*. This mangling is consistent with the Java bytecode language, where a method name is a composite of the name and the types of the arguments. Also, since "`-`" may not appear in a Java name, our convention cannot introduce a collision with any other methods in the source.[3]

[3]Method names do not require further additions to distinguish them from field names, as field names are now sufficiently distinct.

```
abstract class Animal {
 abstract Animal feed();
 abstract Animal feed(int f);
}
```

(a) Overloaded methods in Java

(**define class** *Animal* ⋯
 (**define** (*feed*) ⋯)
 (**define** (*feed-int f*) ⋯)
 ⋯)

(b) Overloaded methods in Scheme

**Figure 5.3**. Compiling overloaded methods

As shown in Figure 5.2, constructors are further identified with special names. The constructor for `Fish` is *Fish˜constructor*, while the constructor for `Cons` from Figure 5.1 is *Cons-int-List˜constructor*.

Scheme classes support both private and public members. A public method is denoted with a **public** form, listing public methods, all identifiers created with a **field** form are public. Anything inside the class declared using the **define** form is private (a **private** form exists for methods). However, a `private` Java member *does not* translate to a **private** Scheme member.

Java `static` members translate into module-level definitions in the Scheme program, which are not part of the class. Therefore, these values do not have access to private members of a Scheme class. Since Java allows static members to access all of the containing classes members, so the compiled members cannot be private. Scheme provides a form, **define-local-member-name**, that makes a class member name local to the current module. By declaring all private fields and methods within this form, the compiler ensures that the names cannot be accessed externally. All accesses within the **module** are checked by the compiler. The compiler does not preserve package and `protected` access restrictions.

### 5.3.4 Nested Classes

In Java, a nested class may either be `static` or an instance class, also known as an inner class. An inner class can appear as a member of a class, within statement blocks, or after `new` (i.e., an anonymous inner class).

Static nested classes are equivalent to top-level classes that have the same scope as

their containing class, with the restriction that they may not contain inner classes. These can be accessed without directly accessing an instance of the containing class. When compiled to Java bytecodes, nested classes are lifted out and result in separate `.class` files. The ProfessorJ compiler also lifts nested classes, and provides a separate module for external access of static nested classes. A nested class and its containing class are members of the same cycle, so both definitions appear in the same module.

Inner classes are also compiled to separate classes. Unlike static nested classes, they may not be accessed except through an instance of their containing class or within the containing method. A separate module is therefore not provided, and construction may only occur through a method within the containing class. The containing class gains one method for each constructor in the inner class.

For member inner classes, the name of a nested class is the concatenation of the containing class's name with the class's own name. For the following classes

```
class Fish {
  class Baby {
  }
}
```

class `Baby` is accessed as `Fish.Baby`. The method added to `Fish` to construct an instance of `Baby` has the name *construct-Baby*, where any arguments for the constructor follow the same convention as previously described. For named classes within a block statement, the class name is further amended with unique information, so that it cannot conflict with other class names when the class is lifted. Similarly, anonymous inner classes are given a unique name, without any additional amending information, and lifted, as is done by bytecode compilers.

Access to the containing class, and any final parameters where appropriate, is provided through an *init-field*. This field is set in the argument list of the call to **new** contained in the constructing method. These fields cannot be accessed externally as no functions are provided.

### 5.3.5 Statements and Expressions

Java and PLT Scheme both enforce the same evaluation order on their programs. Therefore, those Java constructs that are subsumed by Scheme constructs have a straightforward translation. For example,

```
bool a = varA < varB, b = varA > 0;
if (a && b)
 res = varA;
else
 res = varB;
```

translates into

```
(let ((a (< varA varB))
      (b (> varA 0)))
  (if (and a b)
      (set! res varA)
      (set! res varB)))
```

with annotations specifying the source location. Indeed, the majority of Java's statements and expressions translate as expected.

Numbers, characters, and booleans translate directly into Scheme values. Boolean and mathematical operations translate into standard Scheme operations where possible. Thus, unlike the Java specification, numbers do not have a limited range and will automatically become bignums. This is desired behavior in the Beginner and Intermediate languages, where the curriculum does not intend for students to confront overflow. Future versions of the compiler will use operations that cause overflow in the Advanced, Full and Java + dynamic languages (mathematical operations on two dynamic values will not overflow).

Not all Java statements translate directly into Scheme. The primary exceptions are `return`, `break`, `continue`, `finally`, and `switch`, which implement statement jumps. For all of these except `switch` (which is not implemented) and `finally` (which will be discussed in Section 5.5.3), the jumps are implemented with **let/ec**.

The **let/ec** form captures an escape-only continuation in a similar manner to the **let/cc** form:

```
(define-syntax let/cc
  (syntax-rules ()
    ((let/cc k expr ...)
     (call-with-current-continuation
       (lambda (k) expr ...)))))
```

Both forms bind the continuation to the specified variable, and escape back to the point of the **let** with a value.

A `return` translates into an invocation of a continuation that was captured at the beginning of the method. This allows the method to return the specified value, regardless

of the position of the return within the method body. For example, an implementation of the method `feed` within the `Fish` class,

```
Animal feed() {
  return new Fish(weight + 1);
}
```

translates into

```
(define/public feed
  (lambda ()
    (let/cc return-k
      (return-k (make-new-fish (+ weight 1))))))
```

where the *make-new-fish* is a short-hand introduced here for the object creation previously described.

Within a `for`, `while`, or `do` loop, the `break` and `continue` statements terminate and restart the loop respectively. Capturing one continuation outside the loop body implements the functionality of both these features. Although this hinders the readability of the generated program, this is an acceptable trade-off.

For the `while` loop

```
while(true) {
  if (x == 0)
    break;
  else if (x == 5)
    continue;
  x++;
}
```

the translation into Scheme is

```
(let/ec loop-k
  (let loop ((continue? #f))
     (when #t
          (if (= x 0)
             (loop-k (void))
             (when (= x 5)
                  (loop-k (loop #t))))
          (set! x (+ x 1))
          (loop #f)))))
```

The *continue?* argument present here provides valuable information in a `for` loop, for determining when the increment portion of a the loop needs to be rerun. It is included here for consistency.

Capturing a continuation, using either **let/cc** or **let/ec** is an expensive operation in PLT Scheme. While compiler optimizations could remove many of the capturing calls, this

does not overall improve performance. A future version of the compiler will incorporate a Scheme source-to-source optimizer, that eliminates such **let/ec** patterns, putting each statement in a separate **letrec**-bound function and chaining them. This solution not only improves the performance of Java programs, but could provide overall speed benefits to other Scheme programs without sacrificing expressiveness.

### 5.3.6 Native Methods

Most Java implementations use C to provide native support. This system, naturally, uses Scheme as the native language. When the compiler encounters a class using `native` methods, such as

```
class Time {
 static native long getSeconds(long since);
 native long getLifetime();
}
```

the resulting **module** for `Time` **require**s a Scheme module, *Time-native-methods*, that must provide a function for each native method. The name of the native method must be the Scheme version of the name, with *-native* appended at the end. Thus a native function for `getSeconds` should be named *Time-getSeconds-long-native* and `getLifetime` should be *getLifetime-native*.

Within the compiled code, a stub method is generated for each native method in the class, which calls the Scheme native function. When `getSeconds` is called, its argument is passed to *Time-getSeconds-long-native* by the stub, along with the class value, relevant accessors and mutators, and generics for private methods. An instance method, such as `getLifetime`, additionally receives `this` as its first argument.

## 5.4 Compiling for Interoperability

In order to support interoperability with Scheme for Java + `dynamic` , the back-end of the compiler must be aware of `dynamic`, and compile accordingly. Additionally, wrappers must be generated for every class so that instances of the class can be passed into `dynamic` contexts and retrieved from `dynamic` values.

### 5.4.1 Objects and Methods

In order to support `dynamic` objects, each class requires a guard for the possibility that an instance of the class will flow into a dynamic context and each class requires a converting wrapper for the reverse situation. In Chapter 4, an implementation using mirrors

was presented for these guards and assertions. So that Scheme programmers can follow a Scheme convention in calling methods, mirrors are not used in this implementation.

Figure 5.4 presents a Java class and the resulting guard and conversion wrapper generated for instances of the class. The *guard-Fish* class is used to protect an instance of class Fish entering a Scheme function. A guard class exists for each Java class and interface, and extends the guard for the parent class (which provides wrappers for the accessible methods of the parent class). It contains one method for each new method of the class, as well as a method that conforms to Scheme naming conventions where applicable. Similarly, the *convert/check-Fish* class is used to wrap an instance of an object matching the Fish interface coming from Scheme. Before creating this object, a check is performed to ensure that the entering instance contains the correct methods.

Methods in the *guard-Fish* class accept a variable number of arguments to control the error reporting an incorrect number of supplied arguments. Arguments are checked for the correct type, and then converted using a *convert/check*. Results are guarded using a *guard* class before returning. In the *convert/check* classes, methods take the correct number of arguments and check the results.

For the guard, classes contain additional methods whose names follow a Scheme naming convention. The naming convention is described in Section 5.4.6. This feature allows Scheme programmers to call methods on objects that may have originated in Scheme or Java. Overloaded methods do not receive this treatment, so that the Scheme programmer must specify which overloaded method should be used.

### 5.4.2 Fields

When wrapping objects, fields cannot be directly encoded into the wrapped object. The field must always reflect the current value of the objects field, with the appropriate wrappers. Figure 5.5 demonstrates the implementation of a *guard* for a field. In this implementation, the compiled accessor function, part c, dispatches to the accessor method in the wrapper class in circumstances where the given object is not an instance of the correct class. The method in the wrappers strips off one layer of wrapping and attempts again to gain access to the class.

### 5.4.3 Inserting Dynamic Checks

For positions that acquire guards and checks, the compiler back-end queries type information collected by the type-checker and generates the required checks around the

```
class Fish extends Animal {
  String getColor(String format) { ... }
}
```

(a) Java class

```
(define guard-Fish
  (class guard-Animal ()
    (define (fish ···))
    ; ". args" puts all arguments into a list
    (define (getColor-String . args)
      (if (= (length args) 1)
          (if (is-string? (first args))
              (apply-string-guard
               (send fish getColor
                     (apply-string-conversion/check
                      (first args))))
              (raise TYPE-MISMATCH-ERROR))
          (raise ARG-NUMBER-WRONG-ERROR)))
     (define (get-color . args) ···)
     (super-new fish ···)))
```

(b) Scheme Guard Implementation

```
(define convert/check-Fish
  (class convert/check-Animal ()
    (define (fish ···))
     (define (getColor-String arg)
       (let ((res (send fish get-color (apply-string-guard arg))))
         (if (is-string? res)
             (apply-string-conversion/check res)
             (raise TYPE-MISMATCH-ERROR))))
     (super-new fish ···)))
```

(c) Scheme Convert/Check Implementation

**Figure 5.4**. Java class and wrappers for `dynamic`

```
class Fish {
  Food favorite;
}
```

(a) Java class with fields

```
(define guard-Fish ···
  (define (get-favorite)
    (apply-guard-fish (Fish-favorite-get fish)))
  ···)
```

(b) Guard with field

```
(define Fish-favorite-get
  (let ((fish-get ···)) ;Retrieves field accessor from class
    (lambda (obj)
      (cond
        ((is-a? Fish obj) (fish-get obj))
        (else (send obj get-favorite))))))
```

(c) Field accessor function

**Figure 5.5**. Implementing fields in wrappers

value. In the following program, the use of `dynamic` variable `twoFish` must be an `Object`

```
Fish oneFish;
boolean continue( dynamic twoFish ) {
  return oneFish.equals(twoFish);
}
```

Prior to the call to the `equals` method, the compiler inserts a check followed by creating an instance of the *convert/check-Object* class with `twoFish` as the argument, producing the following code

```
(if (check-Object? twoFish)
    (apply-convert/check-Object twoFish ···)
    (TYPE-MISMATCH-ERROR))
```

The function *check-Object* uses a Scheme predicate on objects to confirm that the object contains appropriately named methods for the `Object` class, and that the methods accept at least the specified number of arguments. The actual implementations of the methods may accept more arguments. Instances of the actual `Object` class, as well as instances of `guard-Object`, satisfy this predicate. The ··· in the code are filled with source information for the original location of `twoFish` in the method call.

When the type-information specifies a primitive value, the `dynamic` expression is wrapped with a contract built from the appropriate predicate, such as *integer?* or *char?*. The contract is provided source information for the location of the value.

Additionally, when dynamically checked variables are used as objects, with unknown classes, or functions, the contract check verifies only that the object contain the specific method or field mentioned at the site. The resulting value of the field access or method call acquires its own checking/wrapping.

### 5.4.4 Supporting Equality and Casting

As with any wrapping strategy, testing for equality of objects using `==` becomes more difficult, since the wrapped object is not obviously equivalent to an unwrapped object. This problem is rectified by changing the implementation of `==` from a straightforward translation to *eq?* into a predicate that understands the wrapper system.

The revised `==` first examines the two objects for *eq?*-level equality. If the two objects fail this comparison, and if either of the objects are an instance of a known wrapper, then the predicate queries the wrapper for an equality test. The equality method within the wrapper strips off the current wrapper, checks the unwrapped value for equality with the passed in value, and finally dispatches if the wrapped quantity is itself wrapped. Eventually, the comparison bottoms out with two completely unwrapped values to compare with *eq?*.

Casts incur a similar problem, with an additional twist. Not only does the cast-check within the wrapper need to remove wrappers to determine whether the underlying object is an instance of the specified class (or supports the interface of the class), it must re-wrap the class in the correct wrapper. While the wrapper may not be necessary in all actual cases, it is still necessary in general to ensure that Scheme subclasses of Java classes can be used as the original Java class, even though the subclass may disregard type requirements from the parent class.

### 5.4.5 Inheritance

Scheme programmers can directly subclass Java classes and implement Java interfaces. While the Scheme program may violate type expectations of the methods in the class, introducing an instance of the class into Java through a `dynamic` variable preserves Java's type-expectations.

However, a Java programmer cannot presently easily derive a subclass from a Scheme class. This limitation stems from the inability to extract sufficient information from the Scheme class to distinguish overloaded from overridden methods, initialization argument requirements, and other information. This information is necessary not only to prevent

the Java class from incurring runtime errors but to prevent the generated class from incurring class analysis errors.

The Java programmer can provide this information to the compiler through a `.jinfo` file, which contains sufficient information for Java to use the Scheme class as a Java class. With this information, the Scheme class can be used directly as a Java class, for instantiation, extension and overriding, or instance tests.

One problem remains: every class in Java extends `Object`, but not every Scheme class does so. To resolve this mismatch, the compiler does not actually treat `Object` as a class. Instead:

- The core `Object` methods are implemented in a mixin, *Object-mixin*. Therefore, `Object` methods can be added to any Scheme class that does not already supply them, such as when a non-`Object` Scheme class is used in Java.
- The `Object` class used for instantiation or class extension in Java code is actually (*Object-mixin* object%).
- `Object` instance tests are implemented through an interface, instead of a class. This works because `Object` has no fields (fortunately) so the class is never needed.

A `.jinfo` file indicates whether a Scheme class already extends `Object` or not, so that the compiler can introduce an application of *Object-mixin* as necessary. A Scheme class can explicitly extend a use of *Object-mixin* to override `Object` methods.

### 5.4.6 Naming Conventions

Permitted names in Scheme programs are a superset of permitted names in Java programs. This can cause problems when a Java programmer calls a Scheme function with a name that cannot appear in a Java program (such as *eq?*). While the system could require that Java programmers create a new Scheme library that remaps all of the unsuitable Scheme names into suitable ones, this is onerous in practice.

With standard naming conventions, Scheme programs frequently contain names containing the characters “-”, “?”, and “>”. To ease cross-language references, the Java compiler converts Scheme names that follow certain conventions to names using Java conventions. Dashes in Scheme names are dropped, and the following letter is capitalized; arrows `->` in scheme names are replaced by `To`, and the following letter is capitalized; trailing question marks are replaced by `P`; trailing exclamation marks are replaced by `Set`; and trailing %s are replaced with `Obj`.

Thus, a Scheme function named *slice-fish* is accessed as `sliceFish` within a Java program, and *fish-ok?* as `fishOkP`. Module names also receive this treatment to allow greater accessibility of Scheme from Java. When the Scheme module exports a name that cannot be fit into Java using this convention, the Java programmer can resort to providing a new renaming module. With this naming convention, however, few names will require this treatment.

## 5.5 Java–Scheme Data Mapping

Java provides two kinds of built-in data: primitive values, such as numbers and characters, and instances of predefined classes. The former translate directly into Scheme, and most of the latter (in `java.lang`) can be implemented in Java. For the remainder of the built-in classes, we define classes directly in Scheme.

### 5.5.1 Strings

Although the `String` class can be implemented in Java using an array of `char`s, to facilitate interoperability, `String` is implemented in Scheme. Thus a Scheme string holds the characters of a Java string, and operations on `String`s become operations on strings. The `String` class contains a private field that is the Scheme string and a public method, *get-mzscheme-string* to return an immutable string value.

The *apply-check/convert-String* and *apply-guard-String* functions convert between the two representations. The first function wraps a Scheme string in an instance of the `String` class (instead of a *check/convert-String*), and the latter extracts the Scheme string using *get-mzscheme-string*. Thus, despite the difference in string representations, the data do not need to be converted by either Scheme or Java programmers.

### 5.5.2 Arrays

A Java array cannot be a Scheme vector, because a Java array can be cast to and from `Object` and because assignments to the array indices must be checked (to ensure that only objects of a suitable type are placed into the array). For example, an array created to contain `Fish` objects might be cast to `Object[]`. Assignments into the array must be checked to ensure that only `Fish`, or subclasses of `Fish` such as `Shark` or `Goldfish`, objects appear in the array.

To allow casts and implement Java's restrictions, a Java array is an instance of a class that descends from `Object`. The class is entirely written in Scheme, and array content is

implemented through a private **vector**. Access and mutation to the **vector** are handled by methods that perform the necessary checks.

### 5.5.3 Exceptions

PLT Scheme's exception system behaves much like Java's. A value can be raised as an exception using **raise**, which is like Java's `throw`, and an exception can be caught using **with-handlers**. The **with-handlers** form includes a predicate for the exception and a handler, which is analogous to Java's implicit instance test with `catch` and the body of the `catch` form. The body of a **with-handlers** form corresponds to the body of a `try` before `catch`. Java's `finally` clause is implemented using `dynamic-wind`, which triggers the code when leaving the body of the expression.

Unlike Java's `throw`, the PLT's **raise** accepts any value, not just instances of a throwable. Nevertheless, PLT tools work best when the raised value is an instance of the `exn` record. This record contains fields specifying the message, source location of the error, and tracing information.

When the `Throwable` is given to `throw`, a contract is implied that coerces the Throwable into the Scheme form. A `catch` form implies a contract that expects a Scheme exception record and coerces the value into a `Throwable`. In circumstances where the exception value does not reflect a `Throwable` instance, the coercion creates a new `Throwable` instance of the appropriate type with the information provided by the exception record. Internally, the Scheme exception is embedded in the Java `Throwable` as with `String` and Scheme strings.

Besides generally fostering interoperability, this reuse of PLT Scheme's exception system ensures that Java programs running within DrScheme get source highlighting and stack traces for errors. All of Java's other built-in exception classes derive from Throwable, and therefore inherit this behavior.

## 5.6 Related Work

The J2S compiler [52] compiles Java bytecodes into Scheme to achieve good performance of Java-only programs. This compiler additionally targets Intel X86 with its JBCC addition. J2S globally analyzes and optimizes the bytecode to enhance performance. Java classes compile into vectors containing method tables, where methods are implemented as top-level definitions. Instances of a class are also represented as vectors. Unlike the ProfessorJ system, this compilation model does not facilitate conceptual interoperability

between Scheme and Java programs. Native methods may be written in Scheme, C, C++, or assembly, which allows greater flexibility than with our system at the cost of potential loss of security. As with our system, J2S does not support reflection.

Several Scheme implementations compile to Java (either source or bytecode) [22, 53, 54, 55, 56]. All of these implementations address the interaction between Scheme and Java, but whereas the ProfessorJ compiler must address the problem of handling object-oriented features in Scheme, implementors of Scheme-to-Java implementors must devise means of handling closures, continuations, and other Scheme data within Java:

- JScheme [22, 53] compiles an almost-$R^4RS$ Scheme to Java. Within Scheme, the programmer may use static methods and fields, create instances of classes and access its methods and fields, and implement existing interfaces. Scheme names containing certain characters are interpreted automatically as manglings of Java names. Java's reflection functionality is employed to select (based on the runtime type of the arguments) which method to call. This technique is slower than selecting the method statically, but requires less mangling.

  Accessing Scheme from Java requires the use of an API. The code fragment below illustrates a use of Scheme's complex numbers through the JScheme API:

  ```
  import jscheme.JS;
  ...
  int x, y;
  ...
  JS scheme = new JS();
  scheme.load(new java.io.FileReader("math.init"));
  Object z = scheme.call("make-rectangular",
                         scheme.toObject(x),
                         scheme.toObject(y));
  double m =
     scheme.doubleValue(scheme.call("magnitude", z));
  ...
  ```

  JScheme's interoperability through an API is typical of Java–Scheme implementations, and other implementations that target Java.

- SISC [57] interprets $R^5RS$, with a Java class representing each kind of Scheme value. Closures are represented as Java instances containing an explicit environment. Various SISC methods provide interaction with Java [55]. As with JScheme the user may instantiate Java objects, access methods and fields, and implement an interface. When passing Scheme values into Java programs, they must be converted from Scheme objects into the values expected by Java, and vice versa. To access Scheme

from Java, the interpreter is invoked with appropriate pointers to the Scheme code.

- The Kawa [54] compiler takes $R^5RS$ code to Java bytecode. Functions are represented as classes, and Scheme values are represented by Java implementations. Java static methods may be accessed through a special primitive function class. Values must be converted from Kawa specific representations into values expected by Java. In general, reflection is used to select the method called, but in some cases, the compiler can determine which overloaded method should be called and specifies it statically.
- In addition to a C back end, Bigloo [58, 56] also offers a bytecode back end. For this, functions are compiled into either loops, methods or classes (to support closures). Scheme programmers may access and extend Java classes.

MLj [59] compiles ML to Java, and it supports considerable interoperability between ML and Java code. MLj relies on static checking and shared representations ensure that interoperability is safe; not all ML values can flow into Java. Of course, MLj adds no new degree of dynamic typing to Java.

PLT Scheme developers have worked on embedding other languages in Scheme, including Python [60], OCaml, and Standard ML. None of these implementations supports the level of interoperability between the Scheme language and the compiled language as the ProfessorJ compiler provides with Java.

# CHAPTER 6

# IN THE FIELD

Assessing the benefits and drawbacks of pedagogic languages requires not just observing and analyzing their use in a classroom, but also observing and analyzing a control group that is not using the pedagogic languages. Doing a full psychological study, as such an endeavor would be, could provide information on the overall benefits of pedagogic languages and compilers to assess the potential impact for a tool that allows customized pedagogic languages.

However, doing such a study requires significant efforts on the part of faculty, students, and researchers. At least two distinct (yet similar in curricular content) courses must be run; students in all courses must be observed and participate in specialized assessments, and a random sampling of students from all courses must be followed through some portion of their subsequent careers to assess the impact of their earlier education. Doing such a study has not been feasible.

Instead, when teachers have chosen to use the ProfessorJ languages within their courses, information-gathering surveys have been used to inform the future course of the languages and gain some understanding of the benefits of pedagogic language subsets of professional languages. Due to the lack of a control group, full observation, test-based assessment, or review of progress through subsequent courses, these assessments cannot reach the evaluative abilities of a full study. Nevertheless, the anecdotal evidence gathered is useful for forming an initial assessment of the strengths and weaknesses of language levels and the ProfessorJ languages in particular. It can also be beneficial in suggesting improvements in the languages.

The remainder of this chapter presents the means of gathering information regarding ProfessorJ usage, discussion of the usage of ProfessorJ in a classroom, and presentation of the results of user surveys.

## 6.1 Anecdotal Assessment

The assessment of the ProfessorJ language levels took place through a variety of means to gather anecdotal evidence. These include direct observation of students programming, bug reports indicating troubling error messages, student response and surveys, as well as teacher surveys. These sources provided information regarding the suitability of the language levels to variants of the intended curriculum and the perceived impact of language levels in these courses.

While the ProfessorJ languages were designed to cover a second semester curriculum that builds on the HtDP curriculum, not all of the courses using ProfessorJ have followed this format. It has also been used as the primary compiler in first programming classes, during the last few weeks of a first semester HtDP course, during the second semester after a procedural programming course using Java syntax, and during the second semester of a high school course using an abbreviated HtDP curriculum. In each of these, the same curriculum was followed as much as possible as in the intended course.

## 6.2 ProfessorJ in the Classroom – Self-Evaluation

During the summer of 2005, I presented a first programming course to novices using the ProfessorJ language levels and a combination of the HtDP and follow-up curriculums. The course used all three language levels, spending roughly six weeks of a twelve-week course in the Beginner language, three in Intermediate, one using Advanced, and one using Advanced and Eclipse.

Through teaching this course, I could observe student reactions to the language levels in class, lab, one-on-one sessions, and in their programs. I could also note areas where the language levels were either insufficiently or overly restrictive for the students. At the end of the course, I collected written survey responses from students regarding their opinions of the tool. However, these responses cannot be considered unbiased. The students knew that I had created the software and might be able to associate responses with students, despite my assertions that responses would not affect their grades.

### 6.2.1 Teacher's Perspective

The teaching experience demonstrated certain restrictions in the language that did not fit with the curriculum order. Two examples are `instanceof` and interfaces vs abstract classes.

When presenting object comparison and `equals` using the Intermediate language, the lack of `instanceof` made the resulting programs and explanations awkward and confusion. To implement the body of an equals method for a class with no fields, students and professor alike struggled with either returning true in all instances or raising a class cast error in cases where the passed in object was not equal. This difficulty resulted in adding `instanceof` into Intermediate.

While observing students programming in 2020, I noted that some students tended to use `instanceof` to avoid method dispatch. This motivated delaying the construct into the Advanced language. However, after observing my students' programming patterns using Beginner and Intermediate, I saw no indication that the students would use `instanceof` to avoid method dispatch. With the current curriculum, the benefits of the construct now outweigh the drawbacks of potential abuses.

At the time of the course, the language levels presented `abstract` classes prior to `interfaces`. Students struggled with the differences between the two constructs since, when `interfaces` were introduced, the students were not actually conceptually ready for the notions of multiple inheritance and saw no other benefits to interfaces. This helped to motivate moving the concept earlier to provide more distinctions.

The language levels did provide benefits beyond postponing material in the course. Some students, with minimal programming backgrounds, attempted to write incoherent programs that mutated variables in incorrect ways. Left to their own devices, the students may have found a program that compiled (and produced a correct result on their input). Because of the error messages from the language level, however, the students met with the course staff to discuss a different methodology for solving their problem.

Additionally, I encountered some general problems. For the error messages to be effective, students must understand the words and actually read the error messages. As a young instructor, I occasionally forgot to define technical terms to the students, who then encountered confusion when seeing the term for the first time in an error message. Also, despite the students' lack of experience, they were already habituated not to read error messages. This problem took time to overcome by reading error messages in class to them when I would make a mistake, and by asking them to first read to me the error message when problems arose in their programs.

I did not encounter some problems that I had previously observed in students who were not using language levels. I did not encounter students abandoning a homework

implementation due to the problems in it. Instead, in general, students were able to solve the problems they encountered and move on with the same program. I did not see students blame the language/compiler for errors in logic.

The languages and error messages allowed the students to work within their understanding to solve most syntactic problems, coming to the course staff primarily for higher-level concerns. Other than the problems outlined above, the languages allowed me to present the concepts I wanted, when I wanted, without extraneous language details.

### 6.2.2 Students' Perspective

At the end of the course, students were asked to participate in a user survey regarding ProfessorJ. Of the sixteen students enrolled in the class, only five were present on the day of the survey, so only these students participated. This small sample size lowers the effectiveness of the survey and so this information is augmented with complaints and compliments made by students to the course staff (consisting of myself and a teaching assistant).

**6.2.2.1 Survey Responses.** Of the five respondents, four judged themselves to be complete programing novices and the fifth had minimal previous experience programing in Java. This reflects the makeup of the entire class, where only four or five students had prior programming experience, although the previously known languages also included C and C#. The students were asked questions on the error messages, overall usage of the system, and testing. For the complete survey, see Appendix B.2.

All of the survey respondents found the error messages of ProfessorJ to be generally comprehensible to them. One student indicated that the messages helped track down program errors with greater ease than the error messages with a standard Java compiler (which we transitioned to at the end of the course). However, another student indicated that while the messages were generally comprehensible, occasionally the jargon caused confusion, since the words were unknown. This particular problem was also voiced to the course staff by a few students.

Many of the error messages for ProfessorJ were designed for novice object-oriented programmers, who had some prior programming experience. Because of this expectation, the error messages contain jargon that a complete novice programmer may not know. This is a problem of using the tool with the wrong curriculum, where the error messages need to explain more jargon at the very first. A declarative system that also allows

specification of either the level of error message or text for certain jargon replacement could address this problem.

Of the overall system, the students appreciated the interactivity that the system provided and the coloring and highlighting of the definitions window. However, the students also desired some of the features they encountered in the Eclipse environment, including greater debugging support (for more complicated programs) and auto-name completion.

When asked whether they would prefer to continue using ProfessorJ and the language levels, two students responded in the affirmative. These students desired more time with a simple, student tailored system. The remaining three students either indicated that they were ready to move on or felt that practice in a professional environment would benefit them more.

Finally, the survey also asked about testing preferences. These students used two testing solutions throughout the course. In the first, the students inserted graphical boxes into the definitions in which they could enter their tests. If a test succeeded, the box displayed a green check. A red 'x' appeared if the test failed. Additionally, students used a jUnit-like testing library, where the students extended a test class and provided test methods.

Overall, the students preferred the graphical interface to the textual due to the simplicity and the visual feedback. Despite the connections to a professional library, the students found the nongraphical solution too complex for their first testing experiences.

**6.2.2.2 Personal Comments.** Throughout the semester, students commented to me and my assistant regarding their experiences using the ProfessorJ compiler. The majority of the complaints came further into the semester, as the programs became more difficult, the software became less thoroughly tested, and possibly as the students became more comfortable with us. Compliments came from several of the students throughout the course, both from the more experienced students and novices.

As the programs became more complex, students desired a means of investigating the intermethod behavior. The lack of debugging or stepping support bothered them. Additionally, the lack of libraries other than the drawing library caught their attention. Towards the end of the semester, the students desired more complicated programs and would have preferred interacting with more libraries. These complaints can be addressed with further development work.

From their statements, some students liked receiving only one error message at a time, as this permitted them to focus on the problem. They found that the error messages guided them to correcting the problem quickly, when they could understand the language of the error message. Others mentioned that, unlike previous courses, in early programs there were fewer potential problems to uncover when debugging a program and the interactions window allowed them to debug more quickly.

### 6.2.3 Conclusions

From this experience, it seems that the language levels do provide benefit to the instructor and the students. However, the language levels need to be tailored not just to the curriculum but also to the specific knowledge level of the students. Thus, not only should the declarative specification support adding checks and removing constructs, it should also provide a means of tailoring the vocabulary of the error messages to the specific course.

## 6.3 ProfessorJ in the Classroom – External Evaluation

As previously mentioned, the ProfessorJ compiler has been used in several high school and university classes. Instructors for some of these courses volunteered to answer a user survey regarding the experience. And after one course, students were given a usage survey to assess their experience with the language levels.

### 6.3.1 Teachers' Perspective

Seven instructors volunteered to complete the user survey found in Appendix B.3. The request for feedback was sent to a mailing list for individuals who had completed a training course in using ProfessorJ and teaching the accompanying curriculum, as well as individuals who were known to have used the compiler in their course. Survey participants were not anonymous. People who may have used the compiler and left the mailing list did not receive a request to participate. Therefore, the survey respondents should be considered a friendly audience who may not reflect all experiences.

Two of the seven respondents used ProfessorJ in high school courses prior to covering AP material, while the others used the compiler in university courses – in either the first or second semester. Only one respondent used ProfessorJ as the first programming tool in the first course. Primarily, the students had the intended experience level of one prior

semester programming experience. All respondents followed the intended curriculum as closely as possible. One respondent used only the Beginner programming level before moving to a different environment; the others used Beginner and Intermediate, with one allowing students to use Advanced if they chose. Table 6.1 enumerates the schools for the survey participants, what course(s) the participant taught, the current and previous textbooks for the course, as well as the experience level of the students in the course.

Six respondents indicated that using ProfessorJ's language levels improved their ability to present material. The remaining respondent indicated that the tool benefited

**Table 6.1**. Teachers Using ProfessorJ: N/R indicates no response, N/A indicates the teacher had not taught the course before, B indicates novice students, I indicates intermediate students. U.N.A.M is the Universidad Nacional Autónoma de México.

| School | Course | Exp. | Textbook |
|---|---|---|---|
| U. of Utah | Intro. to C.S. I | B/I | With ProfJ: HtDP & HtDC<br>Before: N/A<br>With Tool: N/A |
| Northeastern U. | Fund. of C.S. 2 | I | With ProfJ: HtDC<br>Before: Object of Data Abs. & Strct.<br>With Tool: JPT & Metroworks |
| Knox College | Intro. to C.S. | B | With ProfJ: HtDC<br>Before: Objects First with Java<br>With Tool: BlueJ |
| Colby College | Data Struc. & Alg. | I | With ProfJ: HtDC & Goodrich&Tomassia<br>Before: Standish<br>With Tool: N/R |
| Spackenkill HS | Programming II | I | With ProfJ: HtDC<br>Before: Software Solutions<br>With TOol: BlueJ |
| U. N. A. M. | Intro. Prog. | all | With ProfJ: N/R<br>Before: N/R<br>With Tool: BlueJ |
| | Data Struct. | all | With ProfJ: N/R<br>Before: N/R<br>With Tool: BlueJ |
| | A.I. | all | With ProfJ: N/R<br>Before: N/R<br>With Tool: BlueJ |
| Owatonna HS | C.S. 2 | I | With ProfJ: N/R<br>Before: N/R<br>With Tool: N/R |

instruction, but did not elaborate. Of those who saw improvement, three indicated that the language levels allowed them to focus only on the syntax relevant to the current concept, which allowed them to spend greater time conveying the important information. Others indicated that the interactive (and textual) nature of the environment allowed them to clearly show their students the effects of expressions and statements.

One respondent, using the compiler in the first course, did indicate that the language levels were not completely suited to their curricular needs. In the Beginning language level, the `if` statement does not permit the different branches to be enclosed in {}, which this respondent found confused his students. Further, the bitwise expression operator caused confusion and incorrect program behavior. For the intermediate level, the instructor would prefer to teach about static and overloaded methods instead of delaying.

Based on the experiences of two of the respondents, several language changes have been made to make the languages provide stronger support for the intended curriculum. These include moving `abstract` classes and inheritance from the Beginning level; restricting `==` to compare only integral numbers, booleans, and characters; and many vocabulary changes to the error messages.

All of the respondents indicated that using the compiler had a positive impact on their students. One of the high school teachers presented the second semester of his pre-AP class using the compiler, and now has students from this program in his AP class. These students show a greater understanding of Java programming and object-oriented concepts than students who had taken the previous pre-AP course.

Four respondents indicated that the language level's restrictions assisted the students in distinguishing instance variables from method parameters and other local variables. And many noted that their students had fewer problems with error messages, although there were still problems.

Overall, the respondents indicated that there are areas for improvement in the compiler. For some, the language levels do not quite fit their curricular requirements. For all, the vocabulary of the error messages needs to continue to be refined for the knowledge level of the students. However, the restrictions do provide them the ability to focus their lectures on the material they desire to and spend less time explaining error messages. All respondents indicated they would use ProfessorJ again.

### 6.3.2 Students' Perspective

In the fall semester of 2003, the first semester course at the University of Utah taught the HtDP curriculum for two-thirds of the semester, followed by the HtDC curriculum using an early version of the ProfessorJ compiler. Before switching to Java, the students learned functional programming in Scheme followed by studying mutation. In Java, the course covered the Beginning language level, including inheritance, method dispatch, and recursion, and the Intermediate level, including object polymorphism and a mutation in an object-oriented setting. At the end of the course, students were asked to participate in an anonymous survey, in Appendix B.1.

This course used the first released version of ProfessorJ, which contained many bugs and environmental integration problems. Exposure to these bugs and usage problems negatively colored some students' experience with the system.

Of the approximately 100 students taking the course, 89 participated in the survey. The course consisted of students with no prior programming experience and students who had written (self-described) large programs in the past. The students can opt out of the course by passing the AP exam. Twenty-six percent of survey respondents had no prior programming experience, thirty-nine percent had written small programs, twenty-four medium sized, and eleven percent had written large programs.

Table 6.2 presents the overall assessment of the students' opinions about the experience. This opinion combines the reaction to the curriculum, the tool, and, for a few students, the course as a whole. Overall, the students indicated that the experience was beneficial to them. The students with the most experience indicated that while the environment did little to assist them, it lowered the amount of help they gave classmates compared to previous courses. Many students who fall into the mixed reaction (neither overall favorable or unfavorable) complained about the bugs encountered when using the software.

A few students, across the board, found the language levels too restrictive and not sufficiently applicable to professional programmers. These students desired that the course and the tool be geared towards teaching them to use the Java libraries and other professional tools instead of the presented curriculum. At the time, no libraries were available to the students.

Other students enjoyed either the language levels or the tailored error messages (some students who disliked the levels still appreciated the error messages). These students

**Table 6.2**. Students' Assessment of Overall Experience

| All respondents | |
|---|---|
| Favorable experience | 52% |
| Mixed experience | 29% |
| Unfavorable experience | 12% |
| Indeterminable | 7% |
| **No prior experience** | |
| Favorable experience | 52% |
| Mixed experience | 35% |
| Unfavorable experience | 9% |
| Indeterminable | 4% |
| **Written small programs** | |
| Favorable experience | 53% |
| Mixed experience | 26% |
| Unfavorable experience | 12% |
| Indeterminable | 9% |
| **Written medium to large programs** | |
| Favorable experience | 48% |
| Mixed experience | 29% |
| Unfavorable experience | 16% |
| Indeterminable | 6% |

indicated that the language levels allowed the professor to concentrate on the interesting portions of programming, and that the levels allowed the students to ease into Java. One student indicated that the language levels lowered their trepidation about learning a new professional programming language.

Students with prior experience indicated that overall the error messages were easy to understand, while those with less experience found some messages confusing. Specific confusion arose with parsing error messages that did not provide sufficient contextual information for the student.

In addition to general impressions about the overall experience and language levels, students were given the opportunity to list features of the system that either helped them or caused them problems. Table 6.3 presents a break-down of the features that students found beneficial, while Table 6.4 presents a break-down of items that caused the students problems.

For both questions, students could freely respond with any number of items. The tables present those answers given by more than one student, where some responses have been placed into specific categories. For example, if a student enumerated several different

**Table 6.3**. Beneficial Features of ProfessorJ, broken down by prior experience. Percentages are percent of entire group, students could select multiple features or no features.

| Beneficial Feature | Percent |
|---|---|
| All respondents | |
| Familiar interface | 16% |
| Error messages | 14% |
| Interaction window | 15% |
| Language levels | 3% |
| No prior experience | |
| Familiar interface | 4% |
| Error messages | 17% |
| Interaction window | 4% |
| Language levels | 4% |
| Written small programs | |
| Familiar interface | 21% |
| Error messages | 18% |
| Interaction window | 15% |
| Language levels | 6% |
| Written medium to large programs | |
| Familiar interface | 19% |
| Error messages | 7% |
| Interaction window | 19% |
| Language levels | 0% |

**Table 6.4**. Problematic Aspects and Features of ProfessorJ. Percentages are percent of entire group; students could indicate multiple aspects, or none.

| Problematic aspect/feature | Percent |
|---|---|
| All respondents | |
| Compiler bugs | 45% |
| Test suite | 28% |
| Error messages | 12% |
| Parse Error messages | 1% |
| Textual representation | 2% |
| No prior experience | |
| Compiler bugs | 52% |
| Test suite | 17% |
| Error messages | 4% |
| Parse Error messages | 0% |
| Textual representation | 0% |
| Written small programs | |
| Compiler bugs | 47% |
| Test suite | 38% |
| Error messages | 12% |
| Parse Error messages | 0% |
| Textual representation | 3% |
| Written medium to large programs | |
| Compiler bugs | 39% |
| Test suite | 26% |
| Error messages | 19% |
| Parse Error messages | 3% |
| Textual representation | 3% |

forms of software bug, the answer was counted as simply compiler bugs. The percentages shown in the tables are the percentage of the entire group that wrote the item.

As is evident from Table 6.4, the test suite caused students considerable problems. At the time of the course, testing was carried out by inserting graphical boxes into the definitions window. The interface for this feature was also in development and students encountered many bugs. Further, they indicated that they found the interface to use this feature cumbersome and that it did not provide sufficient feed back.

Error messages appear in both tables. Many students indicated both that they liked the error messages and that the error messages caused them difficulty. Some clarified that the vocabulary of certain error messages could cause them considerable confusion even though overall the specificity of error messages assisted them in correcting problems. Additionally, some students indicated that when internal errors arose, the internal error

message did not help them in correcting their program. When this was explicitly stated, the complaint was classified as disliking the bugs in the system. However, it is possible that some respondents who did not clarify had more problems with bugs instead of the intended error messages.

The survey additionally asked whether the student would like to continue using ProfessorJ or advance to a professional compiler supporting full Java. Twenty-nine percent of respondents indicated they would like to continue using ProfessorJ. The common reason for this preference was to gain more familiarity with Java in a familiar environment with tailored error messages. The common reason to move on was either that the respondent felt ready for the full language or they thought that experience with a professional tool would benefit them more. Eighteen percent of respondents did not answer this question.

### 6.3.3 Conclusions

From the teachers' experience reports, it can be deduced that language levels do provide the ability to focus on high level concepts instead of syntactic issues. However, even for teachers using a very similar curriculum (with variations based on the particular requirements of individual schools) the language levels can require adjustment to properly present the topics. Also, error messages presently in the system do not always use the correct vocabulary for introductory students.

For the students, more access to professional libraries may quell some concerns over not learning with the professional environment. However, the small increments of the language levels does allow the novice to focus on one concept at a time instead of feeling overwhelmed with syntax required to make the program work.

## 6.4 Developing Libraries for ProfessorJ

As mentioned in Section 6.3.2, some students would like to write programs using multiple libraries, which are not provided by the ProfessorJ compiler. External instructors have recognized this desire and attempted to correct the deficiency for their classes.

At least two developers have attempted to create individualized libraries for the ProfessorJ system using the interfaces available prior to the release of the Java + `dynamic` language. In the surveys from instructors, one respondent mentioned that the older interface was too difficult to accommodate his desire to augment the existing library with ones that met his institution's needs.

Despite working closely with other PLT developers and myself, one of the attempted implementors encountered several pitfalls due to the nature of the connection between Java and Scheme. The library attempted to link a Java drawing interface to the Scheme drawing libraries, similar to the library presented in Chapter 3. However, unlike the presented library, this interface presented a deep class hierarchy with different drawing capabilities.

The base class of the hierarchies contained native methods to connect to the Scheme library. These native methods accessed a noninstance field for the canvas, as the library implementor did not include an instance field whose type would never be correct. This led the developer to include all of the native methods' functionality within one module to access the hidden canvas. This choice led to the native method implementation requiring access to the subclasses, which led to a nonobvious illegal dependency cycle which the implementor could neither understand nor see how to resolve without direct support. Most instructors do not have the available time to work through such problems. In the Java + `dynamic` language, these specific problems are extremely unlikely to occur as the programs do not require unseen levels of indirection and dependency.

While no external programmers have created libraries using the Java + `dynamic` language level, due in part to lack of time, the process of creating a library using the system is easier than for the older system. The two first steps are necessary regardless of any language interoperability details – understand the professional library, and design a suitable interface for students. While in the steps of actually connecting the libraries, an implementor would previously have needed to understand the native interface and dependency requirements between Java and Scheme files, now the implementor can reason about a Java program that accesses the professional library with known types (`dynamic`).

Connecting the library to the DrScheme environment does require one additional step, that Scheme programmers (and Java programmers in Java only systems) do not have. The ProfessorJ compiler must be explicitly informed, in an installation file, which Java files to compile (and their order to ensure proper compilation). This requirement can be eliminated with extensions to the ProfessorJ compiler.

## 6.5 Onwards

While language levels provide educational benefits for instructors and students, no single set of pedagogic levels is sufficient across all courses. Even instructors using similar

curricula experience incongruities between the language supported by the compiler and the language needed in class that require modifying classroom presentation. Although the existing language level implementation could be tweaked, manually tweaking for even one specific incarnation of a curriculum is time-consuming, no matter the programmer's level of expertise.

Instead, a declarative specification of languages could be tweaked by nonexperts to allow the presented languages to mirror the languages presented in class. Building a customizable system will entail designing a declarative specification language, development of new parser technologies, as well as modifying the existing type-checking infrastructure.

### 6.5.1 Specification Language

Pedagogic languages may require modifications to the Java type-system, restrictions of language features based on context (i.e., assignment only permitted within constructors), or additions to the Java language. Other pedagogic languages may simply remove constructs without modifying any static restrictions. A graphical configuration system may be desirable for simple specifications.

For simple restrictions, the language specification should be correspondingly simple, for both specifying the constructs within the language and specifying simple constraints on the constructs. For example constraints like the following should be simple to specify: that constructors are not only permitted but are required; fields are permitted and they must be initialized. Additionally, error message vocabulary should be easily constrained (i.e., mapping concepts, such as class instantiation, to the desired terminology).

More complicated modifications to the professional language, including additions to the type requirements and additions to the language, will require more control within the specification. A textual representation of the specification will be desired for more complicated languages.

The simplest specification could contain a list of language constructs present within the language (where the default constructs contain a canonical set of names). Operators to specify contextual restrictions — such as "only one", "required", "only within" — could be applied to the construct within the list. Modifications to the type system could be applied as functions provided to similar operators, including a "remove-check" operator. Operators to control overloading, naming, and scope could be applied within an entire class, method, or the language overall.

In adding new constructs, the language implementor must provide base parsing, type-checking, and compilation information. Additionally, the implementor must extend the specification language with the name of the new construct and potentially new operators. These operations could be performed by special operators. New features could be added as traditional macros — taking new Java to traditional Java — or by providing a new function to compile the new construct directly to the target language. In the former case, the type-checker must check the added construct with the provided analysis, and then check the transformed Java code as necessary. Any errors occuring in the transformed code should be reported as errors with the language script and not the student's program.

For any specification, the system must perform as many checks as possible to ensure that the language grammar is satisfiable (and at least warn of ambiguities) and consistent. If possible, the system should also analyze the satisfiability of the type specifications. A small amount of internal analysis will be necessary; for example, if the language removes type-checking but supports type-based overloading, no language should be built from the specification.

Language specifications should be able to build upon existing specifications, both to aid in reuse when building subsets of the language and so that instructors can perform simple adjustments to an existing language. Two operators should be provided for this, "include-from" and "exclude-from". If a language specification is extended without these, then the new language should contain all of the features and restrictions of the old language with any constructs included in the new specification.

### 6.5.2 Separating Error Messages

The present compiler contains two kinds of error messages: intentional error messages caused by erroneous student programs, and internal errors reporting a fault within the compiler. The new system must contain three kinds of errors: the two existing kinds, and an error arising from undetected inconsistencies in the language specification. For example, if a built-in check requires information from a different check (such as method argument types), then the compiler will be unable to preform appropriately if the language configuration does not include both checks. As much as possible, these situations should be detected, during compilation where possible and at runtime if necessary, and reported as errors in the specification.

To aid in the debugging of specifications, the resulting error message should include as much source information regarding the specification as possible. Source information could

be removed during an optimization pass if the language specifier indicates. Unfortunately, errors in the language specification may arise as errors in the students' program when the specification does not meet the designer's intention; this is unavoidable.

### 6.5.3 Parsing

The parsing system must generate parsers by integrating (without human intervention) different productions into one parser, with error messages that specify the point of error and the form of the correct construct with vocabulary suited to students. Additionally, the system ought to account for similarities in different language constructs in both parsing and error reporting.

As discussed in Chapter 5, the ProfessorJ parser for error messages contains productions for incorrect language constructs that follow common behaviors observed in students. Additionally, some parse error messages for syntactically similar constructs, such as field and method declarations or casts and parenthesized expressions, refer to both constructs in the message and arise from grammar productions that may match the prefix for either production. These productions support error messages that point out the erroneous portion, note the syntactic similarity between the constructs, and suggest modifications for either syntactic form.

Existing parser generators can be leveraged to build parsers from a specification, support error message development, or to a limited extent both. However, on their own, none of the existing systems adequately supports both requirements of modular extensible parser generation and student-sufficient error reporting.

Existing work on generating parse errors falls into two categories: error recovery, where the parser attempts to correct the parse error and continue; and error specification, where error messages arise from implementor specifications (not necessarily hand-written error messages). In the former category, the HLP [61] attempts to provide student friendly error messages based on automatic corrections to the program. This technique appears to falter when syntactic elements are similar, and it can skip over errors. The Merr system [62], a member of the latter category, generates error messages based on examples of correct and incorrect syntax, attempting to match student programs to the examples and provide error messages based on these. This system does seem to provide specified and tuned error messages, but would require significant work for a language implementor to create examples based on the constructs included in each language. While these systems do

not provide adequate support, they may provide a starting place for the error generation system.

Other existing systems may provide strong starting points for modular parser generation, although their current support for error messages is lacking. These include ASF+SDF [63], Eli [64], and IPG [65]. Both ASF+SDF and Eli support defining pieces of a language, including syntactic entities and associated actions, and then combining them into one language processor. It may be possible to use the combination mechanism to include production specific error messages as associated actions into the process of reading the text of the program, but the system may require extension to support this. The IPG system supports incremental generation of parsers based on existing parsers; such a system may be amenable to the creation of parsers that accept different portions of a language by starting with a minimal core and adding productions as constructs are added to the language. Attaching appropriate error messages is still a concern.

Another parser generator does attempt to combine modular parser creation with error message support. The antlr parser generator [66] supports specifying a parser by selecting different productions, represented by class-like entities. Parser productions can be extended through inheritance of these class-like entities. Error messages can be associated with the different productions to be pieced together. This technique is insufficiently strong, as it does not support the ability to specify error messages for syntactic situations that are close to other syntactic features without providing a specific production for these cases.

Other possibilities for producing a suitable parser builder include modifications to a Yacc/recursive-descent joint parser as is used in ProfessorJ, or exploring natural language processing approaches to parsing. In the former situation, individual language productions might be connected to functions that provide different potential error situations and corresponding error messages. These functions would require many pieces of external information to be supplied based on the other pieces of the language. This approach may not provide support for differentiating within the error message syntactic similarities between language constructs, as discussed earlier.

Natural language processing routinely parses documents that do not have a strict formal structure guiding the parsers. Leveraging these techniques with a programming language may lead to parsers that can suggest multiple modifications to the program resulting in a correct parse. Further, it may be possible to augment these modifications

with explanations (based on the grammar) of how the suggested edit changes the meaning of the program. This style of example modifications, instead of traditional error messages, may be a viable approach for parsing errors for students.

### 6.5.4 Type Checking and Compiling

Unlike parsing, type-checking and compilation can extend the current implementation for arbitrary new languages. The extended type-system must conditionally turn off checks required in the full language as well as accept and run new checks, and the extended back-end must accept translations for new constructs and potentially replace existing translations.

Instead of current hardcoded, guarded analysis, the updated compiler should contain hooks to call analysis functions indicated or provided by the specification. These should be functions, returning no values, that preform a check and signal an error in the event of a violated restriction. The provided input will vary based on the construct that contains the restriction, including relevant local and class-wide information. The mechanism to raise an error in the appropriate context and style will be provided through an error library.

Modifying the behavior of analyses that must provide information, such as method call resolution, will require providing a function that does return the appropriate information. These analyses should be guarded by a condition that switches between the default analysis and the optionally provided new function. If the provided function does not produce information in the correct format, an error blaming the specification should occur.

So that language writers do not have to encode all analysis functions, a library of common functions should be provided. This library can additionally serve as examples of how to create new analysis functions for the various language constructs. This library should be available for extension as well as reuse, so that language designers can easily create libraries containing the analysis functions needed for their languages.

To support adding a new language construct, all analysis and translation functions must support the ability to recognize new abstract syntax nodes and to accept new functions over these nodes.

**6.5.4.1 Error Messages.** The vocabulary of error messages must reflect both the constructs in the language and the words chosen by the specification. Both situations can be handled with parameterized error message functions. Additionally, the specification

language may need to include the ability to swap a built-in error message function for another, in the event that the error message text cannot be sufficiently controlled through vocabulary parameters. If this is necessary, the new error message function should accept similar information to new analysis functions and produce the error message string.

**6.5.4.2 Modifying the Back-end.** As mentioned previously, the system should support two forms of extending and modifying the output of the compiler – a macro-like system and the ability to replace translation functions.

The Java-to-Java transformations would type-check in the full language, with the error messages replaced with internal errors to indicate a problem with the transformation. User-level error messages must be provided by the script author, but could use existing functionality where applicable.

For the second system, only some functions would be replaceable. Functions necessary to the execution of a compiled program (i.e., functions that produce modules) could not be replaced.

# APPENDIX A

# DRAWING LIBRARY

## A.1 Native Interface

### A.1.1 Image

#### Image.java

```
package draw2Native;

public class Image {
  private Object theImage;

  Image( Object i ) {
    this.theImage = i;
  }

  public native Image movePinhole( Posn p );
  public native Image putPinhole( Posn p );
  public native Image overlay( Image i );
  public native Image overlayXY( Image i, Posn p);
  public native Posn getPinhole();
  public native boolean inside( Image isInside );
  public native Posn find( Image inside );
  public native Image addLine(Posn start, Posn end, Color c);
  public native int width();
  public native int height();

}
```

### Image-native-methods.scm

```
(module Image-native-methods mzscheme

  (require (lib "image.ss" "htdp")
           (lib "graphics.ss" "graphics")
           (lib "class.ss")
           (lib "Throwable.ss" "profj" "libs" "java" "lang")
           (lib "RuntimeException.ss" "profj" "libs" "java" "lang"))

  (provide (all-defined-except new-image Image Posn-x Posn-y new-posn Posn))

  ;Necessary
  (define Image (dynamic-require '(lib "Image.ss" "htdch" "draw2Native")
                                 'Image))

  ;Fixed by adding a require-java form
  (define Posn-x (dynamic-require '(lib "Posn.ss" "htdch" "draw2Native")
                                  'Posn-x-get))
  (define Posn-y (dynamic-require '(lib "Posn.ss" "htdch" "draw2Native")
                                  'Posn-y-get))
  (define Posn (dynamic-require '(lib "Posn.ss" "htdch" "draw2Native")
                                'Posn))

  ;new-image: image → Image
  (define (new-image i)
    (let ((new-i (make-object Image)))
      (send new-i Image~constructor-java.lang.Object i)
      new-i))

  ;new-posn: int int → Posn
  (define (new-posn x y)
    (let ((new-p (make-object Posn)))
      (send new-p Posn~constructor-int-int x y)
      new-p))

  ;check: ( any → bool) any (any → void)
  (define (check expected? value error)
    (if (expected? value)
        value
        (error value)))

  ;make-java-runtime-exception: string → RuntimeException
  (define (make-java-runtime-exception str)
    (create-java-exception
     RuntimeException
     str
     (lambda (e s)
       (send e RuntimeException~constructor-java.lang.String s))
     (current-continuation-marks)))

  (define (movePinhole-draw2Native.Posn-native this getters setters
                                               privates posn)
```

```
  (new-image (move-pinhole ((hash-table-get getters 'theImage) this)
                           (Posn-x posn)
                           (Posn-y posn))))

(define (putPinhole-draw2Native.Posn-native this getters setters
                                            privates posn)
  (new-image (put-pinhole ((hash-table-get getters 'theImage) this)
                          (Posn-x posn)
                          (Posn-y posn))))

(define (overlay-draw2Native.Image-native this getters setters
                                          privates image)
  (new-image (overlay ((hash-table-get getters 'theImage) this)
                      ((hash-table-get getters 'theImage) image))))

(define (overlayXY-draw2Native.Image-draw2Native.Posn-native
         this getters setters privates image posn)
  (new-image (overlay/xy ((hash-table-get getters 'theImage) this)
                         ((hash-table-get getters 'theImage) image)
                         (Posn-x posn)
                         (Posn-y posn))))

(define (getPinhole-native this getters setters privates)
  (let ((the-image (((hash-table-get getters 'theImage) this)))
        (checker
         (lambda (val)
           (check
            integer? val
            (lambda (v)
              (raise
               (make-java-runtime-exception
                (format "In class Image, getPinhole expected int given ~a"
                        v))))))))
    (new-posn (checker (pinhole-x the-image))
              (checker (pinhole-y the-image)))))

(define (inside-draw2Native.Image-native this getters setters privates image)
  (check boolean?
         (image-inside? ((hash-table-get getters 'theImage) this)
                        ((hash-table-get getters 'theImage) image))
         (lambda (v)
           (raise
            (make-java-runtime-exception
             (format
              "In class Image, inside expexted to return a boolean, given ~a"
              v))))))

(define (find-draw2Native.Image-native this getters setters privates image)
  (let ((s-posn (find-image ((hash-table-get getters 'theImage) this)
                            ((hash-table-get getters 'theImage) image)))
        (checker
         (lambda (val)
           (check
```

```
              integer?
              val
              (lambda (v)
                (raise
                 (make-java-runtime-exception
                  (format "In class Image, find expected int given ~a" v))))))))
      (new-posn (checker (posn-x s-posn)) (checker (posn-y s-posn)))))

(define (addLine-draw2Native.Posn-draw2Native.Posn-draw2Native.Color-native
          this getters setters privates posn1 posn2 c)
   (new-image (add-line
                 ((hash-table-get getters 'theImage) this)
                 (Posn-x posn1) (Posn-y posn1)
                 (Posn-x posn2) (Posn-y posn2)
                 (send (send c toString) get-mzscheme-string))))

(define (width-native this getters setters privates)
   (check integer?
          (image-width ((hash-table-get getters 'theImage) this))
          (lambda (v)
            (raise
             (make-java-runtime-exception
              (format
               "In class Image, width expected to return an int, given ~a"
               v))))))

(define (height-native this getters setters privates)
   (check integer?
          (image-height ((hash-table-get getters 'theImage) this))
          (lambda (v)
            (raise
             (make-java-runtime-exception
              (format
               "In class Image, height expected to return an int, given ~a"
               v))))))

)
```

### A.1.2 View

#### View.java

```
package draw2Native;

public class View {

  static int count = 0;
  Object display;
  boolean visible = false;
  String name;
  private boolean image = true;

  public View() {
     count += 1;
     name = "View-"+count;
     visible = false;
  }

  public View hide() {
    this.visible = false;
    toggleVisible(visible);
    return this;
  }
  public View show() {
    this.visible = true;
    toggleVisible(visible);
    return this;
  }

  public Image draw( Command c) {
    drawToCanvas(c);
    doubleBufferCall();
    return getBufferCopy();
  }

  public Image drawSequence( CommandSequence commands ) {
    commands.drawAll(this);
    doubleBufferCall();
    return getBufferCopy();
  }

  void allowImage(boolean ok) { image = ok; }

  public native View display( int width, int height );
  native void drawToCanvas( Command c );
  private native Image getBufferCopy();
  private native void doubleBufferCall();
  private native void toggleVisible(boolean v);

}
```

### View-native-methods.scm

```
(module View-native-methods mzscheme

  (require
   (lib "class.ss")
   (lib "mred.ss" "mred")
   (lib "String.ss" "profj" "libs" "java" "lang")
   (lib "Throwable.ss" "profj" "libs" "java" "lang")
   (lib "RuntimeException.ss" "profj" "libs" "java" "lang")
   (lib "Image.ss" "htdch" "draw2Native"))

  (provide display-int-int-native drawToCanvas-draw2Native.Command-native
           getBufferCopy-native doubleBufferCall-native
           toggleVisible-boolean-native)

  (define-struct view (buffer dc canvas frame))

  (define call-back-canvas%
    (class canvas%
      (define call-back-proc (lambda (a) (void)))
      (define/override (on-char char)
        (call-back-proc (to-string char)))
      (define/public (set-callback proc)
        (set! call-back-proc proc))
      (super-instantiate ())))

  (define (to-string ke)
    (let ((ke (send ke get-key-code)))
      (if (char? ke) (string ke) (symbol->string ke))))

  (define (make-java-runtime-exception str)
    (create-java-exception
     RuntimeException
     str
     (lambda (e s)
       (send e RuntimeException~constructor-java.lang.String s))
     (current-continuation-marks)))

  (define (display-int-int-native this field-accs field-sets privates x y)

     ;Fields and field setters
     (let ((visible ((hash-table-get field-accs 'visible) this))
           (set-visible (hash-table-get field-sets 'visible))
           (name ((hash-table-get field-accs 'name) this))
           (get-display (hash-table-get field-accs 'display))
           (set-display (hash-table-get field-sets 'display)))

       (when visible
         (send this hide))

     (let* ((buffer (make-object bitmap-dc% (make-object bitmap% x y)))
            (call-back
```

```
            (lambda (canvas dc)
              (send dc draw-bitmap
                    (send (view-buffer (get-display this)) get-bitmap) 0 0)))
           (frame (make-object frame%
                    (send name get-mzscheme-string) #f (+ x 10) (+ y 15)))
           (canvas (make-object call-back-canvas% frame null call-back))
           (dc (send canvas get-dc))
           (display (make-view buffer dc canvas frame)))
      (set-display this display)
      (send dc clear)
      (send buffer clear)
      (send frame show #t)
      (set-visible this #t)
      this)))

(define (drawToCanvas-draw2Native.Command-native this field-accs
                                                 field-sets privates c)
  ;Fields and field getters
  (let ((visible ((hash-table-get field-accs 'visible) this))
        (get-display (hash-table-get field-accs 'display)))

    (unless visible
      (raise
       (make-java-runtime-exception
        (make-java-string "View must be displayed in order to draw in it"))))
    (send c issue-java.lang.Object (view-buffer (get-display this)))))

;Note: any Scheme program can call these private methods

(define (getBufferCopy-native this field-accs field-sets privates)
  ;Fields
  (let ((image ((hash-table-get field-accs 'image) this))
        (buffer (view-buffer ((hash-table-get field-accs 'display) this))))

    (if image
        (let* ((buffer-bitmap (send buffer get-bitmap))
               (bitmap-copy (make-object bitmap% (send buffer-bitmap get-width)
                              (send buffer-bitmap get-height)))
               (dc-copy (make-object bitmap-dc% bitmap-copy)))
          (send dc-copy clear)
          (send dc-copy draw-bitmap buffer-bitmap 0 0)
          (send dc-copy set-bitmap #f)
          (let ((image-obj (make-object Image)))
            (send image-obj Image~constructor-java.lang.Object
                  (make-object image-snip% bitmap-copy #f))
            image-obj))
        (make-object Image))))

(define (doubleBufferCall-native this field-accs field-sets privates)
  ;Fields
  (let ((dc (view-dc ((hash-table-get field-accs 'display) this)))
        (buffer (view-buffer ((hash-table-get field-accs 'display) this))))
    (send dc draw-bitmap (send buffer get-bitmap) 0 0)))
```

```
(define (toggleVisible-boolean-native this field-accs field-sets privates v)

  ;field
  (let ((frame (view-frame ((hash-table-get field-accs 'display) this))))

    (send frame show v)))

)
```

### A.1.3 GameWorld

#### GameWorld.java

```
package draw2Native;

public abstract class GameWorld extends World {

  Object timer;
  World nextWorld = this;

  public GameWorld() {
    super(new View());
  }

  public native World endOfWorld();


  //Produces a World that will animate with a clock tick of rate
  public final native boolean animate( int width, int height, int rate );

}
```

### GameWorld-native-methods.scm

```
(module GameWorld-native-methods mzscheme

  (require (lib "mred.ss" "mred")
           (lib "class.ss")
           (lib "String.ss" "profj" "libs" "java" "lang")
           (lib "Throwable.ss" "profj" "libs" "java" "lang")
           (lib "RuntimeException.ss" "profj" "libs" "java" "lang"))

  (provide (all-defined-except make-java-runtime-exception))

  (define (make-java-runtime-exception str)
    (create-java-exception
     RuntimeException
     str
     (lambda (e s)
       (send e RuntimeException~constructor-java.lang.String s))
     (current-continuation-marks)))

  (define (animate-int-int-int-native this field-accs field-sets
                                      privates x y rate)

    ;fields
    (let* ((timer-set (hash-table-get field-sets 'timer))
           (timer-get (hash-table-get field-accs 'timer))
           (nextWorld-get (hash-table-get field-accs 'nextWorld))
           (nextWorld-set (hash-table-get field-sets 'nextWorld))
           (get-display (hash-table-get field-accs 'display))
           (set-display (hash-table-get field-sets 'display))
           (display (get-display this)))

      (let* ((draw-sequence
              (lambda (old new)
                (timer-set new (timer-get old))
                (set-display new (get-display old))
                (send (get-display this) allowImage #f)
                (send old erase)
                (send new draw)
                (send (get-display this) allowImage #t)))
             (timer-callback
              (lambda ()
                (let* ((world (nextWorld-get this))
                       (new-world (send world onTick)))
                  (draw-sequence world new-world)
                  (nextWorld-set this new-world))))
             (key-callback
              (lambda (key)
                (unless (string? key)
                  (raise
                   (make-java-runtime-exception
                    (make-java-string
                     (format
```

```
                "Internal error: key must be a string for callback, given ~a"
                key)))))
            (let* ((world (nextWorld-get this))
                   (new-world
                    (send world onKey-java.lang.String
                          (make-java-string key))))
              (draw-sequence world new-world)
              (nextWorld-set this new-world))))
         (timer (make-object timer% timer-callback)))
    (send display display x y)
    (send display keyCallBack key-callback)
    (send timer start rate #f)
    #t)))

(define (endOfWorld-native this field-accs field-gets privates)

  ;fields
  (let ((timer ((hash-table-get field-accs 'timer) this)))

    (send timer stop)
    this))

)
```

## A.2 Java + dynamic

### A.2.1 Image

```
package graphics;

import scheme.lib.htdp.image;
import scheme.lib.graphics.graphics;
import scheme.lib.htdch.graphics.rename;

public class Image {
  dynamic theImage;

  Image( dynamic i ) {
    this.theImage = i;
  }

  dynamic getBitmap() {
    return theImage.getBitmap();
  }

  public Image movePinhole( Posn p ) {
    return new Image(image.movePinhole(theImage, p.x, p.y));
  }

  public Image putPinhole( Posn p ) {
    return new Image(image.putPinhole(theImage, p.x, p.y));
  }

  public Image overlay( Image i ) {
    return new Image(image.overlay(theImage, i.theImage));
  }

  public Image overlayXY( Image i, Posn p) {
    return new Image(rename.overlayXY(theImage, p.x, p.y, i.theImage));
  }

  public Posn getPinhole() {
    return new Posn(image.pinholeX(theImage),image.pinholeY(theImage));
  }

  public boolean inside( Image isInside ) {
    return image.imageInsideP(theImage, isInside.theImage);
  }

  public Posn find( Image inside ) {
    dynamic position = image.findImage(theImage, inside.theImage);
    return new Posn(graphics.posnX(position), graphics.posnY(position));
  }

  public Image addLine(Posn start, Posn end, Color c) {
    return new Image(image.addLine(theImage, start.x, start.y,
   end.x, end.y, c.toString()));
  }
```

```
  public int width() {
    return image.imageWidth( theImage );
  }

  public int height() {
    return image.imageHeight( theImage );
  }

  public boolean equals(Object o) {
     return (o instanceof Image) &&
            rename.imageEqP(theImage,((Image) o).theImage);
  }

}
```

### A.2.2 View

```
package graphics;

import scheme.lib.mred.mred;
import scheme.lib.htdch.graphics.rename;

public class View {

  private static int viewCount = 0;

  private dynamic frame;
  private dynamic canvas;
  private dynamic dc;

  private dynamic buffer;

  private String name;
  private boolean visible = false;

  private boolean image = true;

  public View() {
    viewCount += 1;
    this.name = "View-"+viewCount;
  }

  //Produces a View with a visible canvas of size x and y
  public View display( int x, int y) {
    if (visible)
      this.hide();

    buffer = rename.newObject(mred.bitmapDcObj,
      rename.newObject(mred.bitmapObj,x,y));
    buffer.clear();

    class CanvasInner {
      public void callBack(dynamic canvas, dynamic dc) {
dc.drawBitmap( View.this.buffer.getBitmap(), 0,0);
      }
    }

    frame = rename.newObject(mred.frameObj, name, false, x+15, y+20);
    canvas = rename.newObject(rename.callBackCanvasObj, frame,
           rename.innerToFunction( 2, new CanvasInner()));
    dc = canvas.getDc();

    this.clear();

    frame.show(true);
    visible = true;
    return this;
  }
```

```
//Produces a View without a visible canvas
public View hide() {
  this.visible = false;
  frame.show(visible);
  return this;
}

public View show() {
  this.visible = true;
  frame.show(visible);
  return this;
}

//The Image is a equality testable version of the canvas
//after the drawing command
public Image draw( Image i) {
  drawToCanvas(i);
  dc.drawBitmap(buffer.getBitmap(), 0 ,0);
  return getBufferCopy();
}

//The Image is again an equality testable version of the canvas
//after applying all commands. Issues the commands in reverse order
public Image drawSequence( CommandSequence commands ) {
  commands.drawAll(this);
  dc.drawBitmap(buffer.getBitmap(), 0 ,0);
  return getBufferCopy();
}

void allowImage( boolean ok ) {
  this.image = ok;
}

private Image getBufferCopy() {
  if (image) {
    dynamic bufferBitmap = buffer.getBitmap();
    dynamic bitmapCopy = rename.newObject(mred.bitmapObj,
                                          bufferBitmap.getWidth(),
                                          bufferBitmap.getHeight());
    dynamic dcCopy = rename.newObject(mred.bitmapDcObj, bitmapCopy);
    dcCopy.clear();
    dcCopy.drawBitmap(bufferBitmap, 0, 0);
    dcCopy.setBitmap(false);
    return new Image(rename.newObject(mred.imageSnipObj, bitmapCopy, false));
  } else {
    return new Image(false);
  }
}

void keyCallBack(dynamic curWorld) {
  canvas.setCallback(curWorld);
```

```
  }

  //erases the canvas
  void clear() {
    buffer.clear();
    dc.drawBitmap(buffer.getBitmap(), 0 ,0);
  }

  void drawToCanvas( Command c) {
    if (!visible)
      throw new RuntimeException("View must be displayed to draw in it");
    c.issue(buffer);
   }
}
```

### A.2.3 GameWorld

```
package graphics;

import scheme.lib.htdch.graphics.rename;
import scheme.lib.mred.mred;

public abstract class GameWorld extends World {

  dynamic timer;

  public GameWorld() {
    super(new View());
  }

  public World endOfWorld() {
     timer.stop();
     return this;
  }

  World nextWorld = this;

  void oneStepPrivate(World oldWorld, World newWorld) {
     ((GameWorld) newWorld).timer = ((GameWorld) oldWorld).timer;
     newWorld.display = oldWorld.display;
     display.allowImage(false);
     newWorld.draw();
     display.allowImage(true);
  }

  //Produces a World that will animate with a clock tick of rate
  public final boolean animate( int width, int height, int rate ) {

   class TimerCallBack {
      public void callBack() {
         World old = GameWorld.this.nextWorld;
         GameWorld.this.nextWorld = GameWorld.this.nextWorld.onTick();
         GameWorld.this.oneStepPrivate(old, GameWorld.this.nextWorld);
      }
   }

   class KeyCallBack {
     public void callBack(String key) {
        World old = GameWorld.this.nextWorld;
        GameWorld.this.nextWorld = GameWorld.this.nextWorld.onKey(key);
        GameWorld.this.oneStepPrivate(old,GameWorld.this.nextWorld);
     }
   }

    display.display(width, height);
    dynamic tCB = new TimerCallBack();

    display.keyCallBack(rename.innerToFunction(1, new KeyCallBack()));
```

```
    timer = rename.newObject(mred.timerObj,
                             rename.innerToFunction(0,tCB));
    timer.start(rate, false);
    return true;
  }

}
```

# APPENDIX B

# USER SURVEYS

## B.1 Survey of Students in CpSc 2010 Fall 2003

**ProfessorJ User Survey Fall 2003**

1. What programming experience did you have prior to 2010?
   - None
   - Had written 1 or 2 small programs
   - Had written many small programs
   - Had written medium sized programs
   - Had written large programs
2. If you had programming experience prior to 2010, in what language(s)?
3. How much experience had you had with Java prior to its presentation in 2010?
   - Had never written a Java program
   - Had written 1 or 2 small programs in Java
   - Had written many small programs in Java
   - Had written medium sized programs in Java
   - Had written large programs in Java
4. What was your level of knowledge of Object-oriented programming prior to its presentation?
   - Had never seen object-oriented programming
   - Had heard of but did not fully understand object-oriented programming
   - Fully understood object-oriented programming concepts
5. What is your present level of understanding of OO programming?
   - Do not understand OO
   - Can write programs in OO style, but some concepts still unclear
   - Fully understand OO
6. If you do not feel that you fully understand OO, what areas are still confusing?
7. What aspect of ProfessorJ caused you the most difficulty?
8. What, if anything, did you like about using ProfessorJ?
9. If you could remove 1 feature from ProfessorJ, what would it be?

10. If you could add 1 feature to ProfessorJ, what would it be?
11. What is your overall opinion of the ProfessorJ language levels?
12. If given the choice, would you rather continue using ProfessorJ language levels as you gain familiarity with Java or move to using full Java?
13. If you would like to continue using ProfessorJ language levels, why?
14. If you would like to move to using full Java, why?

## B.2 Survey of Students in CSPP 50101 Summer 2005

**ProfessorJ Usage Survey Summer 2005**

**Prior knowledge**

Please indicate your level of knowledge prior to taking this course:

- Complete novice
- Some knowledge of programming other than Java (or C#)
- Some knowledge of programming in Java (or C#)

If you had programmed in Java before, please list what tools you had used.

**Working with ProfessorJ**

What aspects of using ProfessorJ did you like?

What aspects of using ProfessorJ did you not like?

Did you find that error messages in ProfessorJ were generally comprehensible to you?

**Testing**

Please indicate which style of testing you prefered

- Graphical (minus file corruption woes)
- Extending the Test class with specific test methods

What changes would you suggest for your preferred testing mechanism (minus correcting file corruption)

**The Future**

What feature(s) would you like to see added to ProfessorJ?

Would you prefer to continue using ProfessorJ? Why?

## B.3 Survey of Teachers Spring 2006

**ProfessorJ Teacher Survey Spring 2006**

1. What course(s) have you used ProfessorJ in?
2. What is the average experience level of your students?
   - Beginners (Never written a program to written a few independently)
   - Intermediate (Completed one or two CS courses)
   - Advanced (Completed several CS courses)
3. If students had completed at least one programming course before, please describe the prerequisites of your course (including programming language of prior course(s)).
4. What tools had you used/curriculum had you followed previously?
5. What version(s) of ProfessorJ have you used?
6. What language levels of ProfessorJ did you use?
7. Did you use ProfessorJ in combination with any other tool? If yes, what?
8. How did interacting with ProfessorJ affect your students? (aided, hindered, etc)
9. How did using ProfessorJ affect your instruction?
10. Were the language levels appropriate divisions to your curricular needs? What changes would you make?
11. Did you notice any improvements/deteriorations in student understanding and course progression from before using ProfessorJ?
12. Do you foresee yourself using ProfessorJ in future classes? Why?
13. If you would not use ProfessorJ, what changes in ProfessorJ might affect your decision?
14. How would you rate the overall experience of using ProfessorJ in your class?

# REFERENCES


[1] Holt, R.C., Wortman, D.B., Barnard, D.T.: SP/k: A System for Teaching Computer Programming. Communications of the ACM Vol. 20(5) (1977) pp. 301–309

[2] Findler, R.B., Flanagan, C., Flatt, M., Krishnamurthi, S., Felleisen, M.: DrScheme: A Pedagogic Programming Environment for Scheme. In: Proceedings of the International Symposium on Programming Languages: Implementations, Logics, and Programs, London, UK, Springer-Verlag (1997) pp. 369–388

[3] Kölling, M., Quig, B., Patterson, A., Rosenberg, J.: The BlueJ System and its Pedagogy. In: Proceedings of the Workshop on Pedagogies and Tools for Assimilating Object Oriented Concepts. (2001)

[4] Allen, E., Cartwright, R., Stoler, B.: DrJava: A Lightweight Pedagogic Environment for Java. In: Proceedings of the SIGCSE Technical Symposium on Computer Science Education, New York, NY, USA, ACM Press (2001) pp. 137–141 `www.drjava.org`.

[5] Gray, K.E., Flatt, M.: ProfessorJ: a Gradual Introduction to Java Through Language Levels. In: Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, New York, NY, USA, ACM Press (2003) pp. 170–177

[6] Felleisen, M., Findler, R.B., Flatt, M., Krishnamurthi, S.: How to Design Programs. The MIT Press, Cambridge, MA (2001) `www.htdp.org/`.

[7] Kölling, M., Rosenberg, J.: BlueJ. (2000) `www.bluej.org`.

[8] Conway, R., Constable, R.: PL/CS — A Disciplined Subset of PL/I. Technical Report TR 76-293, Cornell University (1976)

[9] Teitelbaum, T., Reps, T.: The Cornell Program Synthesizer: A Syntax-Directed Programming Environment. Communications of the ACM Vol. 24(9) (1981) pp. 563–573

[10] Wirth, N.: Pascal-S: A Subset and its Implementation. In: Pascal — The Language and its Implementation, John Wiley (1981) pp. 199–259

[11] Ruckert, M., Halpern, R.: Educational C. In: Proceedings of the SIGCSE Technical Symposium on Computer Science Education, New York, NY, USA, ACM Press (1993) pp. 6–9

[12] Hsia, J.I., Simpson, E., Smith, D., Cartwright, R.: Taming Java for the Classroom. In: Proceedings of the SIGCSE Technical Symposium on Computer Science Education, New York, NY, USA, ACM Press (2005) pp. 327–331

[13] Hristova, M., Misra, A., Rutter, M., Mercuri, R.: Identifying and Correcting Java Programming Errors for Introductory Computer Science Students. In: Proceedings of the SIGCSE Technical Symposium on Computer Science Education, New York, NY, USA, ACM Press (2003)

[14] Sanders, D., Dorn, B.: Jeroo: A Tool for Introducing Object-Oriented Programming. In: Proceedings of the SIGCSE Technical Symposium on Computer Science Education, New York, NY, USA, ACM Press (2003) pp. 201–204

[15] Papert, S.: Mindstorms: Children, Computers, and Powerful Ideas. Basic Books, New York, NY, USA (1980)

[16] UVa User Interface Group: Alice: Rapid Prototyping for Virtual Reality. IEEE Computer Graphics Applications Vol. 15(3) (1995) pp. 8–11

[17] Kahn, K.: ToonTalk — An Animated Programming Environment for Children. In: Proceedings of the National Educational Computing Conference, International Society for Technology in Education (1995)

[18] Goldman, K.J.: A Concepts-First Introduction to Computer Science. In: Proceedings of the SIGCSE Technical Symposium on Computer Science Education, New York, NY, USA, ACM Press (2004)

[19] Guzdial, M., Rose, K.: Squeak: Open Personal Computing and Multimedia. Prentice-Hall (2001)

[20] Hagan, D., Markham, S.: Teaching Java with the BlueJ Environment. In: Proceedings of the Ascilite Conference, Ascilite (2000)

[21] Flatt, M., Findler, R.B.: PLT MrEd: Graphical Toolbox Manual. Technical Report TR97-279, Rice University (1997)

[22] Anderson, K., Hickey, T., Norvig, P.: JScheme User Manual. (2002) `jscheme.sourceforge.net/jscheme/doc/userman.html`.

[23] Findler, R.B., Felleisen, M.: Contracts for Higher-Order Functions. In: Proceedings of the ACM SIGPLAN International Conference on Functional Programming, New York, NY, USA, ACM Press (2002) pp. 48–59

[24] Findler, R.B., Flatt, M., Felleisen, M.: Semantic Casts: Contracts and Structural Subtyping in a Nominal World. In: Proceedings of the European Conference on Object-Oriented Programming. Lecture Notes in Computer Science, London, UK, Springer-Verlag (2004) pp. 364–388

[25] Flatt, M., Krishnamurthi, S., Felleisen, M.: A Programmer's Reduction Semantics for Classes and Mixins. Formal Syntax and Semantics of Java Vol. 1523 (1999) pp. 369–388 Preliminary version appeared in Proceedings of Principles of Programming Languages, 1998. Revised version is Rice University Technical Report TR 97-293, June 1999.

[26] Igarashi, A., Pierce, B., Wadler, P.: Featherweight Java: A Minimal Core Calculus for Java and GJ. In: Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, New York, NY, USA, ACM Press (1999) pp. 396–450

[27] Bracha, G., Ungar, D.: Mirrors: Design Principles for Meta-level Facilities of Object-Oriented Programing Languages. In: Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, New York, NY, USA, ACM Press (2004) pp. 331–344

[28] Bracha, G., Griswold, D.: Strongtalk: Typechecking Smalltalk in a Production Environment. In: Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, New York, NY, USA, ACM Press (1993) pp. 215–230

[29] Goldberg, A., Robson, D.: Smalltalk-80: The Language and its Implementation. Addison-Wesley, Reading, MA (1983)

[30] Cardelli, L.: Amber. In: Combinators and Functional Programming Languages : Thirteenth Spring School of the LITP, Val d'Ajol, France, May 6-10, 1985. Vol. 242. Springer-Verlag (1986)

[31] Abadi, M., Cardelli, L., Pierce, B., Plotkin, G.: Dynamic Typing in a Statically Typed Language. ACM Transactions on Computing Systems Vol. 13(2) (1991) pp. 237–268

[32] Duggan, D.: Dynamic Typing for Distributed Programming in Polymorphic Languages. ACM Transactions on Computing Systems Vol. 21(1) (1999) pp. 11–45

[33] Benton, N.: Embedded Interpreters. Journal of Functional Programming Vol. 15(4) (2005) pp. 503–542

[34] Ramsey, N.: Embedding an Interpreted Language Using Higher-Order Functions and Types. Journal of Functional Programming (To appear) Initial version appeared in ACM SIGPLAN Workshop on Interpreters, Virtual Machines and Emulators, pp. 6-14, June 2003.

[35] Lindholm, T., Yellin, F.: The Java Virtual Machine Specification. Second edn. Addison-Wesley, Boston, MA, USA (1999)

[36] Bracha, G.: Invokedynamic: Wednesday September 28, 2005. Computational Theology: Gilad Bracha's Sun Weblog (2005) `blogs.sun.com/gbracha`.

[37] Weiser, M., Demers, A., Hauser, C.: The Portable Common Runtime Approach to Interoperability. In: Proceedings of the ACM Symposium on Operating Systems Principles, New York, NY, USA, ACM Press (1989)

[38] Liskov, B., Bloom, T., Gifford, D., Scheifler, R., Weihl, W.: Communications in the Mercury System. In: Proceedings of the Hawaii International Conference on Software Track, IEEE Computer Society Press (1988)

[39] Bershad, B., Ching, D.T., Lazowska, E.D., Sanislo, J., Schwartz, M.: A Remote Procedure Call Facility for Interconnecting Heterogeneous Computer Systems. IEEE Transactions on Software Engineering Vol. 13(8) (1987) pp. 880–894

[40] Rogerson, D.: Inside COM. Microsoft Press, Redmond, WA (1997)

[41] OMG: The Common Object Request Broker: Architecture and Specification (1997) Revision 2.0 July 1995, Update July 1996, Object Management Group, formal document 97-02-25 www.omg.org.

[42] Hamilton, J.: Interlanguage Object Sharing with SOM. In: Proceedings of the USENIX Conference on Object-Oriented Technologies. (1996)

[43] Wollrath, A., Riggs, R., Waldo, J.: A Distributed Object Model for the Java System. In: Proceedings of the USENIX Conference on Object-Oriented Technologies. (1996)

[44] Platt, D.S.: Introducing Microsoft .NET. Second edn. Microsoft Press, Redmond, WA, USA (2002)

[45] Hamilton, J.: Language Integration in the Common Language Runtime. SIGPLAN Notices Vol. 38(2) (2003) pp. 19–28

[46] Carlisle, M.C., Sward, R.E., Humphries, J.W.: Weaving Ada 95 into the .net Environment. In: Proceedings of the ACM SIGAda International Conference on Ada, New York, NY, USA, ACM Press (2002) pp. 22–26

[47] Beazley, D.M.: SWIG : An Easy to Use Tool for Integrating Scripting Languages with C and C++. In: Proceedings of the USENIX Tcl/Tk Workshop, USENIX (1996)

[48] Blume, M.: No-Longer-Foreign: Teaching an ML Compiler to Speak C "natively". Electronic Notes in Theoretical Computer Science: BABEL '01, First International Workshop on Multi-Language Infrastructure and Interoperability Vol. 59(1) (2001) pp. 36–52

[49] Furr, M., Foster, J.S.: Checking Type Safety of Foreign Function Calls. Technical Report CS-TR-4627, University of Maryland, Computer Science Department (2004)

[50] Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification. Second edn. Addison-Wesley, Boston, MA, USA (2000)

[51] Flatt, M.: Composable and Compilable Macros: You Want It When? In: Proceedings of the ACM SIGPLAN International Conference on Functional Programming, New York, NY, USA, ACM Press (2002) pp. 72–83

[52] Bergeron, É.: Compilation Statique de Java. Master's thesis, Université de Montréal (2002)

[53] Anderson, K.R., Hickey, T.J., Norvig, P.: SILK — A Playful Blend of Scheme and Java. In: Proceedings of the Workshop on Scheme and Functional Programming, Technical Report TR00-368, Rice University (2000)

[54] Bothner, P.: Kawa: Compiling Scheme to Java. In: Proceedings of the Lisp Users Conference. (1998)

[55] Miller, S.G., Radestock, M.: SISC for Seasoned Schemers. (2003) `sisc.sourceforge.net/manual`.

[56] Serrano, M.: Bigloo: A "practical Scheme compiler" User Manual. (2004) `www-sop.inria.fr/mimosa/fp/Bigloo/doc/bigloo.html`.

[57] Miller, S.G.: SISC: A Complete Scheme Interpreter in Java. sisc.sourceforge.net/sisc.pdf (2003)

[58] Serpette, B.P., Serrano, M.: Compiling Scheme to JVM Bytecode: A Performance Study. In: Proceedings of the ACM SIGPLAN International Conference on Functional Programming, New York, NY, USA, ACM Press (2002) pp. 259–270

[59] Benton, N., Kennedy, A.: Interlanguage Working Without Tears: Blending SML with Java. In: Proceedings of the ACM SIGPLAN International Conference on Functional Programming, New York, NY, USA, ACM Press (1999) pp. 126–137

[60] Meunier, P., Silva, D.: From Python to PLT Scheme. In Proceedings of the Workshop on Scheme and Functional Programming, Technical Report UUCS-03-023, University of Utah, School of Computing (2003)

[61] Sippu, S., Soisalon-Soininen, E.: Practical Error Recovery in LR Parsing. In: Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New York, NY, USA, ACM Press (1982) pp. 177–184

[62] Jeffery, C.L.: Generating LR Syntax Error Messages from Examples. ACM Transactions on Computing Systems Vol. 25(5) (2003) pp. 631–640

[63] Klint, P.: A Meta-Environment for Generating Programming Environments. ACM Transactions on Software Engineering Methodology Vol. 2(2) (1993) pp. 176–201

[64] Gray, R.W., Levi, S.P., Heuring, V.P., Sloane, A.M., Waite, W.M.: Eli: A Complete, Flexible Compiler Construction System. Communications of the ACM Vol. 35(2) (1992) pp. 121–130

[65] Heering, J., Klint, P., Rekers, J.: Incremental Generation of Parsers. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, New York, NY, USA, ACM Press (1989) pp. 179–191

[66] Parr, T.: ANTLR Parser Generator and Translator Generator (2006) www.antlr.org/.