5-2016

An Investigation Into the Generality of a Graphical Representation of Program Code for Source to Source Translation

Stephen Schaub
Clemson University, sschaub@gmail.com

An Investigation Into the Generality of a Graphical Representation of Program Code for Source to Source Translation

---

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Computer Science

---

by
Stephen Schaub
May 2016

Accepted by:
Brian A. Malloy, Committee Chair
Jason Hallstrom
Murali Sitaraman
Mark Smotherman

Abstract

This thesis addresses the problem of defining a source-to-source translation system for reusable software components. It describes the development of an interoperable language for writing software components, and presents a system to translate components written in the interoperable language to a set of compatible target languages. The common features in a set of popular programming languages are analyzed to inform the design of the interoperable language. An evaluation is performed by using the source-to-source translator to convert two well-known open source Java libraries to C++ and Python, and the accuracy and performance of the resulting translations are assessed.

Dedication

To my wife, Ruth, for her love and support, and to my father, Paul, for introducing me to the field of computer science.

Acknowledgments

I would like to thank my advisor, Dr. Brian Malloy, for shepherding me through the topic selection, research, and the production of this thesis. I am deeply indebted to him for his encouragement and for the many hours he invested meeting with me, reviewing my work, coauthoring papers, and providing invaluable guidance. I am also grateful to each of my committee members for their helpful feedback on my research: thanks to Dr. Mark Smotherman for introducing me to the department and advising me on my Ph.D. program; to Dr. Murali Sitaraman and the RESOLVE Software Research Group for exposing me to a significant ongoing research project; and to Dr. Jason Hallstrom, whose stimulating classes and seminars provided early writing experience and prepared me for my own research. Finally, I would like to acknowledge the support of my colleagues in the Division of Mathematical Science at Bob Jones University, especially Dr. Gary Guthrie, Dr. Melissa Gardenghi, Dr. Jim Knisely, Mr. Alan Hughes, and Dr. Debbie Summerlin, who have provided encouragement and shouldered additional responsibilities during my leave of absence to help make this research possible.

Table of Contents

List of Tables

List of Figures

List of Listings

Chapter 1

Introduction

The software development landscape has changed radically in its relatively short history. The introduction of portable high-level languages, and in particular, the ascendance of object-oriented technologies, has enabled a degree of software reuse that was unimaginable to the assembly language programmers of the 1950's. And yet, solutions to fundamental problems in software engineering reuse and reliability remain elusive and in many cases unsolved. The proliferation of high-level languages—the mechanism enabling reuse—has been a significant factor in fueling this growth but has also increased the scope of the problem, thereby retarding progress in arriving at solutions.

Despite ongoing efforts, the limitations of existing language interoperability mechanisms continue to inhibit interlingual code reuse. This lack of progress in reuse, in turn, has hindered improvements in software reliability. If there were a single dominant language, or convenient language interoperability mechanisms, there would be strong economic motivation for developing high-quality libraries of carefully specified and formally verified algorithms. The existence of such libraries could increase the reliability of all software development efforts. Because algorithms must be re-implemented in each new language, however, there is currently little incentive for such work.

One approach to interlingual code reuse involves the use of tools that translate the source code of complete programs or libraries from one language to another. Although commercial systems are available for this purpose, they generally suffer from a number of limitations that constrain their usefulness. Terekhov and Verhoef observe that the problem of completely automated source-to-source translation is a difficult one, and that large language conversion efforts are a risky undertaking that have led to bankruptcies and dismantled departments [47]. As popular languages such as

Java and C++ continue to evolve, the resulting feature explosion has exacerbated the problem. In this work, we explore an alternative solution to the challenge of whole language conversion, and demonstrate its utility through the implementation and validation of a system that implements the solution.

Several authors have suggested that building a fully automated source-to-source translation system can be facilitated by defining subsets of languages among which programs can be automatically translated [1, 3, 36]. We believe that the definition of such a subset should be guided by an analysis of a set of popular, high-level languages that share a significant set of semantically compatible features. We assert that an interoperable language whose design is constrained by these compatible features will be rich enough to enable the implementation of useful software components, and that components written in such a language can be automatically translated to compatible languages with good reliability and acceptable efficiency. We acknowledge at the outset that some aspects of languages are commonly left unspecified in language definitions. Thus, no platform- and implementation-independent translation system, automated or otherwise, can hope to achieve complete semantic equivalence. But this limitation should not preclude the exploration of the practical utility of such systems.

The feature explosion that presents a challenge to language translators also presents a challenge for computer science educators. As languages such as C++ and Java have evolved into multi-paradigm behemoths, questions regarding what features to include and what to exclude from introductory programming courses become more difficult to answer. We believe that a curriculum informed by an intersection analysis of high-level languages would naturally emphasize features that are important, and de-emphasize features that are tangential.

Currently, there is no effort described in the literature that facilitates the design and implementation of a source-to-source translation system for enabling software component interoperability that is based upon a carefully specified concrete subset of a high-level language. The existence of such a system would enable the creation of libraries of components that can be translated for use in any of the supported target languages. We therefore propose to create a system for translating components written in an interoperable language into a set of compatible target languages. In order to demonstrate the generality and utility of our approach, our system will include a facility that converts components written in a standard high-level language to our interoperable language.

To facilitate the translation, the system will utilize a graphical representation of the in-

teroperable component source code. Like the interoperable language, this representation will be language-neutral, devoid of any target-language-specific details. The generality of this representation will enable a modular architectural approach that facilitates the enhancement of the system to support additional, compatible target languages. The use of a graphical representation will also enable the use of graphical transformations to facilitate the code generation process.

1.1 Problem Statement

In this thesis, we address the problem of defining a translation system for reusable software components. The problem involves defining a suitable interoperable language for expressing the software components and associated tests, together with a system for automatically translating the components and their tests into a set of target programming languages.

1.2 Research Approach

We begin by identifying a set of popular languages with compatible features. We analyze the common features in the set, and define an interoperable language that implements these features. We build a compiler that translates components written in our interoperable language to a set of target languages. Finally, we validate the compiler using a set of test cases drawn from existing open-source component libraries.

1.2.1 Contributions

Our main contributions are as follows:

- An analysis of a set of object-oriented languages identifying common language features. No comprehensive analyses of this type has been conducted to date.
- A language, and corresponding abstract semantic graph, designed for straightforward translation to the analyzed languages.
- The design and implementation of a tool that translates code from Java to C++ and Python based on our approach.

1.3 Thesis Statement

We assert

1. that several languages currently in wide use share a set of compatible features;
2. that an interoperable language designed around those compatible features is rich enough to specify the implementation of useful software components;
3. that a source-to-source translator can be created to generate accurate translations of components written in an interoperable language into a set of target languages via a fully automatic process; and
4. that the translator can utilize transformations of a language-neutral graphical representation of the source code to facilitate the translation process.

Some terms used in our thesis statement require clarification. A translation will be considered *accurate* if it passes unit tests supplied with the original component. A process that is *fully automatic* requires no human intervention.

1.4 Thesis Organization

We begin with a survey of related work in Chapter 2. Chapter 3 presents background information about program translation, language interoperability, and related topics that are referenced throughout the rest of the paper. Chapter 4 describes challenges and our proposed solutions. Chapter 5 describes our system architecture. Chapter 6 discusses our analysis methodology and the resulting interoperable language. Chapter 7 presents details about our translation to Python and C++. Chapter 8 describes our evaluation methodology and presents the results. Finally, we present conclusions in Section 9.

Chapter 2

Related Work

In this chapter we review the research that relates to source-to-source translation of languages for enabling software component interoperability. We first review research that establishes the usefulness of translating subsets of languages to other languages. We then review approaches to language translation. We also review the literature that describes research for translating from one language to multiple languages, and finally we describe an approach to translating from multiple source languages to multiple target languages.

Source-to-source translation is a well established technique for language interoperability and software reuse. Citing Huijsman et al. [24], Plaisted observes that previous work has shown the value of defining translations on subsets of languages [36]. He argues for the development of abstract languages and subsets of high-level languages, and the creation of translators from those interoperable languages to other high-level languages. Rather than applying translation to large, special-purpose programs, he suggests source-to-source translation is more useful when applied to reusable algorithms, and urges the development of libraries of algorithms in these language subsets.

Many source-to-source systems have been developed that take a single source language as input and translate to a single target language. These translators are special-purpose conversion efforts that are not extensible to other sources or targets. The literature contains numerous examples [24,27,32,49]. Most of these report challenges in performing a completely automated conversion of the entire source language. For example, Trudel et. al created a comprehensive Java to Eiffel translator that successfully ported some large test cases [51]. However, they were unable to translate aspects of Java that are unsupported in Eiffel, such as dynamic loading and weak references.

In another effort [50], Trudel et al. claim support for converting the entire C language as used in practice into Eiffel, with good readability, an impressive claim that is validated with a variety of tests on popular C libraries and applications. However, due to the notoriously unsafe programming practices permitted by C, the authors had to disable Eiffel runtime array bounds checks or make small alterations to source programs in order to successfully translate some of the test cases. Also, they acknowledge that the translation of C `goto` statements into Eiffel, which lacks an unrestricted branch statement, poses one of the biggest challenges to readability. Finally, the Resolve effort [43] is notable for converting verified components to Java.

The level of intermediate representation used by source-to-source translators can affect the readability of the resulting code. Waters observes that most source-to-source translators operate through a process of transliteration and refinement, in which the source program is first converted to the target language, and then optimizations are applied to improve the result [53]. Waters presents an alternative approach termed *abstraction and reimplementation*, in which the source is analyzed and converted to a high-level abstract representation that expresses the essential nature of the overall computation being performed in the original program, and then translated to the target language. Waters describes two systems implementing this approach: Satch [17], a Cobol to Hibol translator, and Cobbler [12], a Pascal to Assembly translator. Waters argues that, when the intermediate representation is at a sufficiently high level of abstraction, the resulting target code will be more idiomatic, and can approach the readability that a human translator would achieve. Waters also notes that this approach, although more difficult to implement and more specialized in nature than transliteration and refinement, can be applied successfully in cases where transliteration and refinement fails. For example, in cases where the source program contains constructs that do not map directly to constructs in the target language, a transliteration system would balk, while a system modeled on Waters' approach would have global information available to select appropriate constructs during the reimplementation phase. However, Waters acknowledges that the abstraction approach is not always practical, due to the requirement that the analysis module recognize the essential computations being performed in the source program. Our approach can be viewed as a low-level application of Waters' abstraction technique, due to the abstract nature of our intermediate representation. However, our abstraction remains close to the level of our target languages, and does not attempt to capture the essential algorithms being translated. Thus, it can be successfully applied in a broader range of cases than Waters', because we do not have the requirement that the analyzer

must recognize the essential algorithms in the source in order to perform a translation.

Relatively few source-to-source systems support multiple output languages generated from a common intermediate representation. Plaisted presents an abstract language, PL, that allows the construction of programs and proofs of correctness, and which is designed to support the translation of programs into multiple concrete languages [35]. However, he does not provide an implementation. Farrow and Yellin outline an approach for a many-to-many source-to-source translation system, and describe their experience implementing a bidirectional translation between Pascal and C [16]. As in our system, the authors design a common intermediate representation for the languages among which they wish to translate, guided in their design by analyzing the similarities and differences in the source and target languages. To perform the translation, the authors devise invertible attribute grammars, which allow them to specify the mapping from Language A to Language B, and then generate the inverse mapping from Language B to Language A automatically. They acknowledge that their system is unable to handle constructs that cannot cleanly map between both languages. They do not evaluate their system as to its completeness or effectiveness.

Albrecht et al. describe a similar effort to translate between subsets of Pascal and Ada [1]. The authors performed an analysis of Pascal and Ada, defining a common subset of both languages using an intersection analysis similar to our approach. They devised a common tree structure capable of representing programs written in the compatible subsets of Pascal and Ada, and a tool set for translating Pascal and Ada programs into the graphical form, and back to Pascal and Ada. The authors observed that, even with languages as closely related as Pascal and Ada, there are subtle semantic differences that create difficulties for defining compatible sublanguages. We view our work as expanding their approach to a larger set of languages with a richer set of features.

Arrighi et al. describe a recent effort to produce a many-to-many translation system, GOOL [3]. Their system is a source-to-source translator for object-oriented languages. The translator supports multiple high-level input and output languages, mapping input programs to an intermediate representation based on an abstract object-oriented language, GOOL, and generating output to the target high-level language from that intermediate representation. The GOOL system is designed to handle a subset of features of its input languages; when it encounters programs that use unsupported features, it passes through the unsupported fragments as comments in the output program. Thus, it is intended as a human-assisted translation system, requiring manual intervention to handle unsupported features, rather than a fully automated system. No language specification is

available for the abstract GOOL language, so we are unable to evaluate the range of functionality their system can support. Also, the authors have not published any reports validating their system, and in our tests, the GOOL system had difficulty handling fundamental constructs such as basic string operations.

We distinguish our work from GOOL in the following ways. First, although both systems make use of an intermediate language that represents a subset of functionality available on popular object-oriented languages, our language provides a specification that defines the supported features. Second, our system supports the fully automatic translation of programs that use features not directly supported by the intermediate language, rather than passing them through as comments to be handled by a manual translation step. Finally, our system explicitly supports the translation of unit tests to be used in validation of the translated code. We note that GOOL claims to support multiple high-level input languages; while we are proposing to support only one high-level input language, there is nothing about our approach that prevents us from supporting multiple high-level input languages.

Several authors have described experience with tools using a graphical transformation approach based on term rewriting. El-Ramly et al. [14] describe their experience implementing a Java to C# translator using the TXL system [10], a functional, rule-based source code analysis and manipulation language designed for building source-to-source transformation tools. TXL performs transformations on the graphical representation of a source program using term rewriting. Like GOOL, the authors' Java–C# translation system is semi-automated, passing unhandled constructs such as nested and anonymous classes through as comments in the translated source. Camacho et al. [8] describe a system for automating the generation of source-to-source translators for spacecraft operations languages using ASF+SDF [4, 22] and the ASF+SDF Meta-Environment [52], another source-to-source tool based on the term rewriting approach to graph transformation. They observe the high degree of similarity between spacecraft operations languages, devise a common intermediate representation, and write annotated grammars to perform the translation mappings. Rather than attempting a comprehensive intermediate representation that accounts for all features in all relevant languages, they design a core IR, and provide an extensibility layer to handle features not in the core through mappings to the core. Finally, Hemel et al. [23] discuss the use of the transformation toolset Stratego/XT [6] in implementing a domain-specific language for generating web applications. Like TXL and ASF+SDF Meta-Environment, Stratego/XT uses a term rewriting ap-

proach to graph transformation. The authors' main contribution is code generation through model transformation, an approach in which the intermediate representation is gradually transformed into target code through the application of a series of declaratively specified tree transformations. In contrast to these declarative term rewriting approaches, our translator system will utilize a visitor-based recursive descent approach to transformation, a standard, often-used technique in translation systems. As a general-purpose traversal algorithm, Visitor [18] is not subject to the constraints of some declarative transformational approaches [31].

Several source-to-source systems have been developed that produce JavaScript as a target language, to facilitate their execution in a web browser [15, 21]. For example, Emscripten [15] is a system developed by Mozilla that translates programs written in the intermediate representation language of the LLVM compiler framework [29] to JavaScript. Combined with the Clang [9] front-ends for C and C++, Emscripten enables complete, fully automated source-to-source translation from C and C++ to JavaScript. The JavaScript translations produced by Emscripten (and similar systems) are intended to be executed in a web browser, and are not suitable to be read and maintained by human programmers.

Some source-to-source systems are designed for language extensibility. In these systems, the input language is a superset of the output language. For example, the Polyglot system [33] allows extensions of the Java language to be developed. Programs written in Polyglot-defined extensions are translated to standard Java programs by a series of AST transformations, and the resulting Java programs can be compiled and run using standard Java tools. Other source-to-source systems permit transformations on program source code. A system of this class is the Rose Compiler Framework [41], which enables transformations on C, C++, and Fortran. Although both Polyglot and the Rose system can handle the full range of features of the language they are designed to extend or transform, they are not designed to emit code in target languages other than the input source language itself.

Chapter 3

Background

This section presents definitions and concepts referenced in the remainder of the thesis. We begin with a survey of key data structures used to represent programs. Next, we survey the types of language families in Section 3.2. Section 3.3 presents a survey of language interoperability approaches. Finally, we survey common programming language features in Section 3.4.

3.1 Program Representation Graphs

Compilers utilize a variety of intermediate representations during the translation of programs, including both graphical and linear forms [44]. Here, we define three related types of graphs commonly used to represent programs in a compiler: parse trees, abstract syntax trees, and abstract semantic graphs.

Given a lexicon (a set of words), a *language* consists of a set of sentences—sequences of words from the lexicon. Languages are defined by grammars, which specify legal sequences of derivation steps that produce the sentences in the language. A *program* is a sentence in a language.

Formally, a grammar is a four-tuple $(N,\ T,\ S,\ P)$, where N and T are disjoint sets of vocabulary symbols, P is a set of rules consisting of sequences of vocabulary symbols from $(N \cup T)$, and S is an element from N termed the *start symbol*.

A *parse tree* is a data structure that results from deriving an input sentence using a grammar G. The root of the tree is the start symbol S in G, and the children of each node in the tree correspond to symbols on the right-hand side of a production P in G.

```
1 class Node {
2   int value;
3   Node next;
4 }
```

Listing 3.1: Source code for class Node

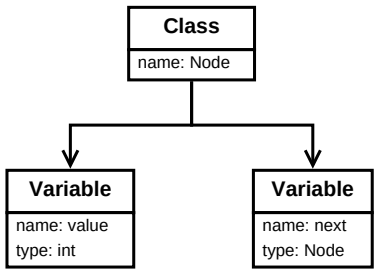

Figure 3.1: A sample Abstract Syntax Tree

An *abstract syntax tree* (AST) is a parse tree that has been transformed through the removal of nodes that do not contribute to the essential meaning of the program, such as those representing nonterminals, punctuation, and keywords.

An *abstract semantic graph* (ASG) is an abstract syntax tree that has been enhanced with additional edges and content that represents semantic information not present in an AST. For example, abstract semantic graphs often contain type information in expression nodes, and edges connecting identifier nodes to declaration nodes.

Listing 3.1 presents a sample Java class definition named Node, containing two fields, an integer value on line 2, and a reference to a Node, next, on line 3.

Figure 3.1 shows a sample Abstract Syntax Tree corresponding to the Node class defined in Listing 3.1. The tree contains a root representing the overall class definition, with two child nodes representing the two fields in the class. The type information for the fields is not yet resolved to definitions.

Figure 3.2 shows a sample Abstract Semantic Graph corresponding to the Node class. The graph represents a transformation of the AST in Figure 3.1, in which the type information for the field nodes has been resolved through the addition of edges labeled *typeDef*. The *value* node now points to a node with additional information about the integer type, and the *next* node has resolved its type definition to point to the node containing the definition of the Node class. This addition of type information has thus transformed the AST into an ASG.

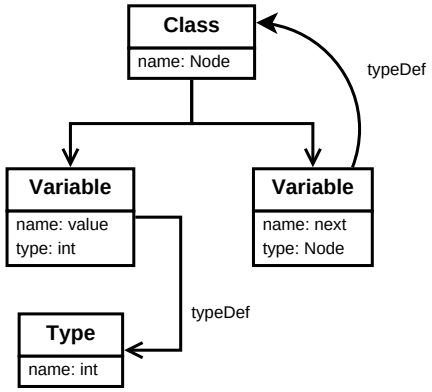

Figure 3.2: A sample Abstract Semantic Graph

3.2 Language Families

Programming languages are commonly characterized as belonging to one or more of the following language families: imperative, functional, object-oriented, and logic. In this section, we briefly survey the characteristics of these families.

Imperative languages take as their computational model the classic Von Neumann architecture, in which programs operate upon data stored in memory by explicitly mutating state. Imperative programs consist of sequences of statements that operate on values stored in variables. Control flow proceeds sequentially except where branching or looping structures modify the sequential flow. Programs are generally organized using the subroutine as the chief organizational unit. Classic examples of these languages include C, Fortran, and Basic.

Object-Oriented languages model information using objects, which are collections of attributes packaged together with the subroutines that manipulate them. They represent an extension to the imperative paradigm, in that explicit mutation remains an important part of the computational model. Programs are generally organized as a set of classes. Java, Smalltalk, and Simula are widely cited examples.

Functional languages de-emphasize explicit mutation in favor of a model in which functions produce values derived exclusively from their input parameters and other variables within scope, without causing side effects. The expression is the basic computational unit; programs consist of functions whose body is generally a single expression, rather than a sequence of statements. Examples of these languages include ML and Haskell.

Logic languages rely on a model of computation based on formal logic, in which a proof

procedure is used to deduce answers to queries. Programs consist of a database of logical statements, including facts and rules for deriving new facts from known facts and rules. Prolog is a well-known example.

Many languages can be characterized as reflecting aspects of two or more of these categories. Examples of such multi-paradigm languages include C++ and OCaml, both of which include a combination of imperative, object-oriented, and functional features.

3.3 Language Interoperability

Language interoperability involves the creation of programs that are written in more than one language. According to Weiser et al. [54], language interoperability typically involves both low-level infrastructure issues such as the sharing of threads, I/O, and a common address space by code in multiple languages; as well as higher-level concerns, such as the ability of languages to invoke each others' subroutines and refer to each others' public identifiers, and the ability to share data representations. A variety of approaches to language interoperability have been explored in the literature. In this section, we present a survey of the approaches.

In one approach to language interoperability, clients in language A access components in language B via *remote procedure calls* (RPC). This model enables a distributed computation, as there is no sharing of address space between the language environments that comprise the overall program. Weiser et al. argue that this approach can work well when the language partitioning logically mirrors the client-server model, although the overhead of remote procedure calls can introduce a significant cost [54].

Another approach to interoperability involves the use of *foreign language interfaces*, which provide a mapping between the calling conventions of language A and language B. Such mappings allow programs in language A to invoke subroutines in language B. This approach is used by tools such as SWIG [46], which automatically generates interfaces for a variety of languages to be able to consume components written in C and C++. This technology is more efficient than the RPC approach, since code for both languages typically resides in the same address space, but some overhead is still to be expected when traversing language boundaries; also, non-idiomatic interfaces can create a readability problem.

A third interoperability technique involves the use of a *common intermediate representation*.

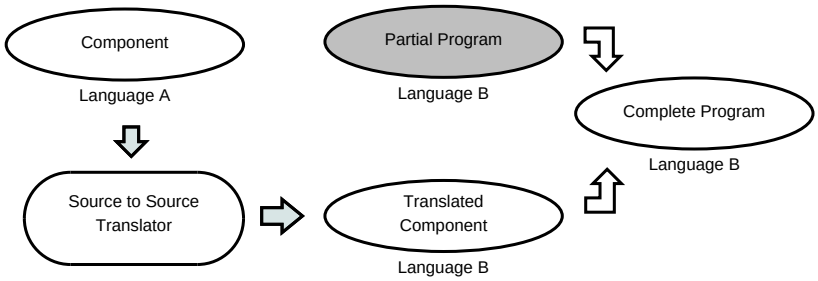

Figure 3.3: Source to Source Interoperability

For example, Microsoft's .NET Framework [37] and the Java Virtual Machine [28] both define an intermediate representation (IR) that supports several source languages. However, these approaches do not support all languages. For example, neither the Microsoft nor the Java IR supports multiple inheritance, thus excluding languages such as C++ that require it. In a variation of this approach, Weiser et al. describe a portable common runtime that acts as a layer between the operating system and language-specific runtime systems, providing features such as memory management and threading support [54]. This approach imposes fewer constraints on the interoperating languages than having a common IR, but has not seen widespread adoption.

Two languages are made interoperable through *source-to-source translation* when a module written in Language A is translated to source code in Language B, so that it can be utilized by a program in Language B. The process is illustrated in Figure 3.3. After the module is translated to Language B, it can be compiled or executed via interpretation in conjunction with the main program written in Language B.

The source-to-source approach to interoperability offers a number of advantages over the others we have described. It can offer greater efficiency, since there is no overhead due to interprocess calls or crossing foreign language interface boundaries, and no need to create a common intermediate representation that necessarily de-emphasizes efficiency to achieve interoperability. If the resulting code preserves good readability, after the component is translated, there is no need to preserve the original in order to maintain the component, yielding improved maintainability. Also, there is increased convenience, as there is no need to maintain two language environments to maintain both the component and the program that uses it.

However, source-to-source translation also poses a number of challenges, which we discuss in Section 4.

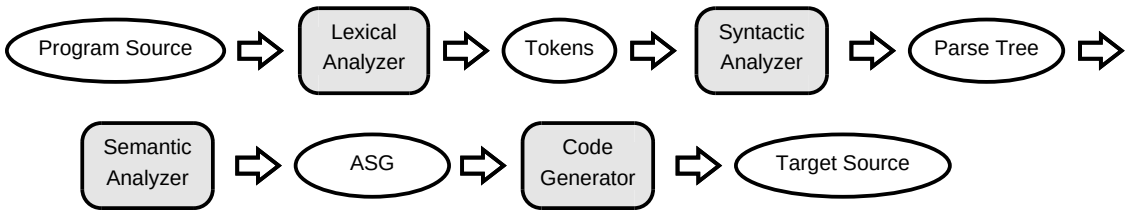

Figure 3.4: Internal Organization of a Source-To-Source Translator

3.4 Language Features

In this section, we survey language features discussed in this thesis. We begin with a survey of features commonly found in imperative languages. Then, Section 3.4.2 discusses object-oriented features.

3.4.1 Imperative Features

In this section, we discuss features commonly found in imperative languages, including data typing issues and parameter passing.

Static vs. Dynamic Typing: One of the most fundamental concerns involving the data types of a programming language is the question of whether variables (and, consequently, expressions) are statically or dynamically typed. In *statically typed* languages such as C, C++, and Object Pascal, the types of all variables must be determined at compile time, either through explicit declaration or type inference. In *dynamically typed* languages such as Python, Ruby, and JavaScript, the types of variables and expressions are unknown until runtime.

Explicit vs. Implicit Typing: Related to the issue of static and dynamic typing is the question of whether data types in variable and function declarations are stated explicitly or are implicit. Dynamically typed languages use *implicit typing*, omitting data types from function and variable definitions. In contrast, statically typed languages utilize *explicit typing*, specifying data types in such definitions. As an alternative to explicit typing, some statically typed languages employ *type inference* to statically determine the type of a variable or function from its context.

Numeric Types: Numeric processing capabilities form a common and important subset of most programming languages. Mirroring the dichotomy found in typical hardware architectures, numeric types and operations are commonly divided into two broad categories: integer and floating point. Other variations, including fixed point and complex numbers, are less common. Statically

typed languages typically provide a variety of both integer and floating point types with varying storage requirements and corresponding ranges. In some languages, the storage requirements and ranges of these types are well-defined; in others, they are unspecified. Some languages provide both signed and unsigned integer types; the unsigned types provide increased range over their signed counterparts. Finally, the handling of errors in numeric operations, including overflow and underflow and division by zero, can vary by language, and within a given language, by implementation.

String and Character Types: Character processing forms another common subset of many languages. Most languages provide a variety of operations on strings, through some combination of library subroutines and language-defined operators. The maximum length of a string is typically implementation defined. Character data in most modern languages is stored using Unicode. Unicode represents a string of characters, called *code points*, using one of several encodings, including UTF-8, UTF-16, and UTF-32. Most languages store character strings in UTF-8 or UTF-16, which are variable-length encodings, requiring multiple *code units* to represent a given code point. String indices in these languages sometimes refer to a code point; other times, to a code unit.

Reference, Pointer, and Value Types: In most languages, some variables hold values, while others store references to values. Variables whose data type is a *value type* hold the values assigned to them, while variables whose type is a *reference type* hold a pointer or reference to a value that is stored elsewhere in memory. In languages such as C and C++, all types are value types, except where a special symbol is appended to the type to indicate that the variable holds a pointer or reference to the actual value. In other languages, such as Java, data types are inherently either value or reference types. For example, in Java, numeric types such as `int` and `double` are value types, while class and array types in Java are reference types.

The distinction between a pointer and a reference is unimportant for our purposes, and in our discussion, unless otherwise indicated, we will use those terms interchangeably.

Parameter Passing Modes: Many languages provide a variety of parameter passing modes for communicating values between the *actual* parameters in the caller of a subroutine and the *formal* parameters in the subroutine interface. The parameter passing modes facilitate passing large objects efficiently, returning values through parameters, and so on. The two most common modes are by-value and by-reference. A *by-value* formal parameter receives a copy of the actual parameter's value, and the procedure is thus unable to modify the corresponding parameter in the caller. A *by-reference* formal parameter receives a reference to the actual parameter, allowing the

procedure to modify the value referenced by the corresponding parameter in the caller.

3.4.2 Object Oriented Features

Object-oriented languages are generally classified into two categories: prototypal and class-based. *Class-based* languages use a class mechanism to define and classify objects: objects are instances of a particular class. In contrast, *prototypal* object systems create new objects by cloning other objects, and do not attempt to classify objects using a class-based hierarchy.

A *class* defines a set of members, including constants, variables, and methods that operate on the class data. Some languages allow type definitions, such as nested classes, interfaces, and function types, to be members of classes. Variables and methods can be instance members or class members. *Instance members* exist within instances of the class, and are accessed through instances; *class members*, also known as static members, exist independently of any instances, and are accessed through the class itself.

Inheritance is used to create specializations of classes in a class hierarchy. *Single inheritance* languages require each class to have at most one direct ancestor, while *multiple inheritance* languages allow a class to have more than one direct ancestor. *Single hierarchy* languages specify a single class as the ultimate ancestor of all other classes; *multiple hierarchy* languages allow multiple class hierarchies, each with its own base class or classes.

Abstract methods are methods that have no implementation. *Abstract classes* are classes that contain at least one abstract method. Abstract classes exist to specify operations that may be overridden by subclasses in the hierarchy. An *interface* is a language feature that consists solely of abstract methods. Single inheritance languages typically allow a class to implement multiple interfaces, thereby gaining some of the benefits of multiple inheritance.

In statically typed languages, a reference variable typed as a base class can be assigned a reference to an instance of a derived class, because the derived class necessarily implements all of the operations defined in the base class. In order to access the additional operations provided by a derived class in this situation, a *checked downcast* operation is performed to allow the derived class instance referenced through the base class-typed variable to be assigned to a derived class-typed variable. A dynamic check is typically performed to ensure that the assignment is legal at runtime.

Virtual methods are methods that can be overridden by a child class, and to which method invocations are dynamically bound. A virtual method can thus exhibit polymorphic behavior when

it is invoked through a reference typed as a base class. Method invocations to *nonvirtual methods* are statically bound, and do not support polymorphism.

Statically typed object-oriented languages typically provide enforced *information hiding* mechanisms for class members, often using visibility modifier keywords. The common modifiers include public, private, and protected. Members marked public can be freely accessed by code outside the class; those marked private may be accessed only by code inside the class. The protected modifier is used in inheritance scenarios to allow child classes to access members in a base class, while protecting against access by classes outside the hierarchy.

Chapter 4

Challenges

In this chapter, we discuss source-to-source translation, its challenges, and some proposed solutions.

4.1 Source-to-Source Challenges

Designers of source-to-source translators have several goals. An effective source-to-source translator must accurately preserve the semantics of the input program, so that the translated version produces the same results as the original when executed. A comprehensive translator will map all constructs in the source language to the target, yielding a fully automated translation. Efficiency of the generated code is also an important consideration. Finally, a good translator will produce a translated program that is readable, ideally yielding highly maintainable code that is idiomatically expressed in the target language.

A number of challenges are involved in applying source-to-source translation to the problem of achieving component interoperability. When source language features are used in the component that are not present in the target language, those features must be emulated in the target; depending on the difficulty of the emulation, the readability and efficiency of the resulting code can suffer greatly. For example, mapping unsafe type operations in languages like Fortran to strongly typed languages like Ada, which prohibit such operations, can be done by modeling the entire memory store using a vector [32]—yielding singularly unsatisfying results.

Even when a particular feature is available in both languages, semantic differences and

undefined behaviors can complicate translation. As an example of the difficulties arising from semantic differences, consider integer overflow. In the C family of languages, integer overflow results in rollover, a behavior on which some programs rely for correct operation. However, in In Ada and Visual Basic, integer overflow triggers an exception. As for undefined behaviors, C and C++ leave numeric data type sizes as an implementation detail, and do not specify the behavior that occurs when array bounds are violated. When such issues are left unspecified by a language specification, producing a reliable translation that is platform- and implementation-independent is impossible.

Finally, translation of standard library calls poses a significant challenge. Ideally, a source-to-source translation should map standard library calls for the source language to the native standard library calls for the target language, rather than introducing an interoperable library layer. However, since standard libraries differ widely in the range of features provided, such a mapping is not possible, in general. For example, standard library facilities used in the source may not be available in the target language, and even when they are, their semantics may differ substantially. In cases when the library source code is unavailable, a completely automated source-to-source translation is impossible.

4.2 Addressing Source-to-Source Challenges

One approach to addressing these issues is to limit the scope of the problem, by restricting source programs to using a subset of language features that form an intersection of the desired target languages. Although this is not a practical solution for tackling the conversion of much legacy code, there are important software reuse scenarios that could be enabled by such a system. As Plaisted observes, general-purpose algorithms such as those used to process data structures have characteristics that differ from code found in custom application programs: they contain relatively few calls to the standard library; tend to be more carefully specified and written; venture less frequently into corner areas of the language that create difficulties for translation; and are frequently reused [36]. Plaisted envisions creating libraries of reusable, general-purpose software components, written in an interoperable subset language, that are designed to be automatically translated to compatible target languages. And, as we demonstrate, such an intersection is rich enough to enable the conversion of legacy libraries that have few standard library dependencies.

We address the problem of translating features in the source language for which there are no equivalent features in the target by careful design of the source language. We restrict the set of

features in the source language to those that can be mapped in a straightforward manner to all of the selected target languages.

To address the problem of semantic differences between languages, we utilize two strategies. In some cases, we leave some behaviors unspecified in our interoperable language. For example, since the behavior of integer overflow differs among our languages, our interoperable language will leave this behavior undefined. In other cases, we will rely on libraries to bridge the semantic gap. For example, our interoperable language will specify automatic memory management semantics; in order to translate programs to target languages that do not support automatic memory management, we will rely on a garbage collection library to supply the needed functionality.

We address the problem of undefined behaviors by ensuring that any behaviors not defined in our target languages are also not defined in our source language. This will ensure that programs cannot rely on a given behavior in the source language that is undefined in the target.

To handle difficulties arising from standard library incompatibility, we define a minimal standard library for our interoperable language. We provide support libraries for our target languages that implement library facilities needed by our test cases that are not present in the target language.

Chapter 5

System Overview

In this chapter, we present an overview of our source-to-source translation system and its architecture. We organize this description into a set of architectural views. We begin with the Functional View, which depicts the primary functionality and high-level organization of the translator. In Section 5.2, we describe the Development View, depicting the package structures and key classes and packages.

5.1 Functional View

Figure 5.1 depicts a high-level view of the translator, which takes as input components, programs, and unit tests written in a subset of Java, and translates them to a set of supported target languages. To allow the validation of our system using test cases that do not conform to the iJava subset of Java, our translator accepts a subset of Java larger than iJava, internally transforming program structures not in the iJava subset into iJava-supported features.

The translator architecture is partitioned into stages to facilitate extensibility to additional target languages, as shown in Figure 5.2. The first stage converts the input Java program to an

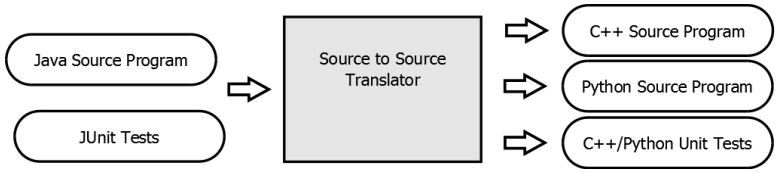

Figure 5.1: System Overview

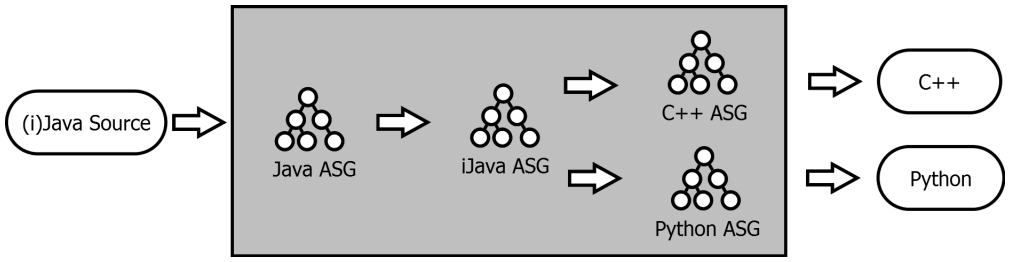

Figure 5.2: Expanded System Overview

abstract representation of Java: an abstract semantic graph (ASG). For this stage, we utilize the high-quality Java parser component in the standard Java compiler, assuring good compatibility with a wide range of Java source programs.

Stage 2 of our system transforms the program encoded in the Java ASG to a language-independent representation encoded as an iJava ASG. Since iJava is a subset of Java, this involves a set of graphical transformations, described in chapter 7. In particular, program structures utilizing Java features not present in iJava, such as for loops and anonymous classes, are transformed to structures that utilize only iJava features.

In **Stage 3** of our system, the iJava ASG is transformed to an ASG in the target language. This involves construction of a graph that closely mirrors the iJava ASG structure, but whose nodes contain data and logic needed for the generation of target code in the final step.

In the final stage, target code is emitted during a traversal of the target language ASG. During this stage, we use Terence Parr's excellent StringTemplate library [45] to construct the target code by extracting information from the target language ASG nodes and inserting it into syntactic fragments defined in template files. This technique represents an application of the well-known Model-View-Controller architectural pattern [18], in which the model is embodied in the ASG nodes, the view is defined by template fragments, and the controller consists of the classes that initiate the code generation.

5.2 Development View

Our system implementation is organized as follows:

- The javaic package contains the JavaParser class, which interfaces with the Java compiler API to produce the Java ASG, and the Main class, the starting point for the translator.

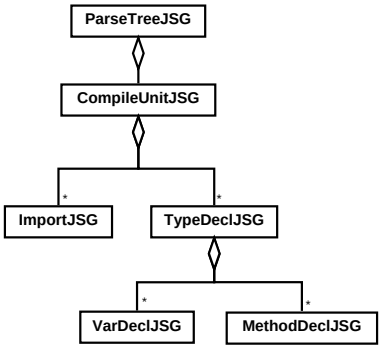

Figure 5.3: iJava ASG Organization (Partial)

- javaic.parsetree and javaic.semantics define the ASG nodes for iJava. Figure 5.3 contains a UML class diagram depicting a portion of the iJava ASG class structure.

- javaic.codegen and its subpackages javaic.codegen.cpp and javaic.codegen.python define the ASG nodes for the target languages, C++ and Python, together with infrastructure for code generation.

The code organization for the back-end of the translator is carefully designed for extensibility, using a combination of the Visitor, Abstract Factory, and Model-View-Controller design patterns [18]. Depicted in Figure 5.4, CodeGenTreeBuilder forms the root of a hierarchy of code generation classes that implement the Visitor design pattern. CodeGenTreeBuilder implements a target language-agnostic traversal algorithm to visit the nodes in the iJava ASG and build an ASG for the target language. The language-specific subclasses of CodeGenTreeBuilder—CPPCGTreeBuilder and PythonCGTreeBuilder—perform graphical transformations required for their respective target languages, as described in Section 7.2. The construction of nodes for the target language ASG is performed by the Abstract Factory implemented in the CGNodeHierarchy class hierarchy, also depicted in Figure 5.4, in which the CppCGNodeFactory and PythonCGNodeFactory create target language-specific nodes for the ASG.

The hierarchy of classes rooted at CodeGenBase contain the shared and target language-specific code that initiates traversal of the iJava ASG to build the target language ASG, passes the resulting ASG as a model to the StringTokenizer library to convert to target code, and finally emits the target code.

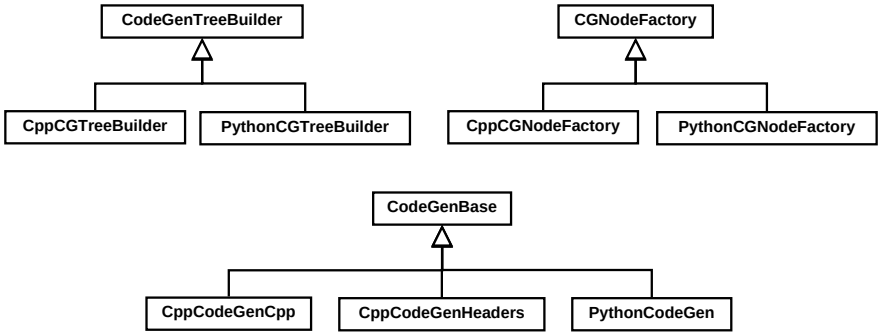

Figure 5.4: Target Code Generation Classes

Chapter 6

Interoperable Language Design

In this chapter, we describe the design of our interoperable language, iJava. We begin in Section 6.1 with a presentation of our analysis and design methodology. Section 6.2 discusses how we selected a set of languages for analysis. Finally, we present the details of our analysis and language design in Section 6.3.

6.1 Design Methodology

In order to determine a set of language features useful for defining an interoperable language, we analyze the intersection of the capabilities of a set of languages. The features included in this intersection determine the definition of our interoperable language, yielding features that can be mapped to suitable capabilities in the analyzed language set.

We agree with Plaisted's observation that interoperable component languages should be imperative languages with side effects, arrays, and records, because of the need for efficiency and applicability [36]. To select the languages, we use the TIOBE index [48], a well-known ranking of programming language popularity based on information provided by web search engines. To increase applicability of our results, we include both statically and dynamically typed languages. We exclude special purpose languages such as those designed primarily for database query and mathematical calculations, because they are not general purpose languages. We also exclude functional languages because of our focus on imperative language capabilities. Finally, since the majority of languages in the top 20 TIOBE include object-oriented features, and including non-OO languages would necessar-

ily exclude important features from our analysis, we narrow our focus to object-oriented languages.

After choosing the languages that we wish to consider, we compare the data types, operators, statements, and structural features of the languages. For a given language feature, we determine the intersection of the capabilities of that feature that are present in each language in our language set. This intersection represents a common subset of functionality for that feature that is found in all of the languages.

In some cases, an intersection of language features may exclude useful or important functionality that can easily be emulated in the languages where it is missing. We adjust the intersection to include the functionality in some of these cases. For example, some languages include both a character sequence type (ex. Java's `String`) and a single character type (ex. Java's `char`), while others omit the single character type. A pure intersection would exclude the single character type, but since having a single character type is a useful feature, and can be easily emulated with a character sequence type, we include both types in our intersection.

In other cases, an intersection may yield undefined semantics. In some of these cases, we choose to define the semantics for iJava. For example, the behavior of division by zero varies among the languages in our set, yielding an empty intersection. We define iJava semantics to throw an exception in this case, because this behavior can be easily implemented using a library function in languages that do not throw an exception for division by zero. In other situations, imposing a definition is not practical, and we must leave the semantics undefined.

As another example of the way we resolve semantic intersections, consider the question of variable typing. Java's variables are statically typed; Python's are dynamically typed. To resolve this dichotomy, we consider which of these two approaches is *more restrictive*. Since statically typed variables are more restrictive than dynamically typed variables, we determine the intersection in this case to be statically typed variables. We observe that mapping programs from a statically typed language to a dynamically typed language requires only the erasure of explicit types from the variable declarations. Mapping code from a dynamically typed language to a statically typed language is difficult or impossible in cases where a variable holds more than one type of value over its lifetime, and, even when possible, leads to abstruse translations.

6.2 Analysis Set Selection

We studied languages in the February 2015 TIOBE top 20 (see Figure 6.1), and selected the following five languages for analysis, following the criteria discussed in Section 6.1:

- **C++** is an important object-oriented systems language that consistently places in the top five in the TIOBE index. The referenced language standard is C++14 [25].
- **Java** is a widely used object-oriented systems language that has consistently ranked in the TIOBE top two. The referenced standard is Java 8 [20].
- **Python** is a popular object-oriented scripting language that has twice earned the distinction of being named language of the year in the TIOBE index. The referenced standard is Python 3.4 [40].
- **JavaScript**, known best for its use in client-side processing for web applications, is an object-oriented scripting language consistently found in the top 15 in the TIOBE index. JavaScript is based on an ECMA standard language named ECMAScript; the referenced standard is ECMAScript 5.1 [13].
- **Object Pascal** is an object-oriented extension to Pascal, one of the most influential academic procedural languages of the late twentieth century. Object Pascal is consistently in the top 20 in the TIOBE index and one of the few that is not in the C family. In contrast to the other languages, there is no official language standard. We analyzed the language implementation Free Pascal, version 2.6.4 [34].

6.3 Language Analysis and Design

Rather than design a completely new language, we chose a subset of an existing language, Java, as the basis for our interoperable language. As our analysis shows, in most cases, a subset of Java's features often falls neatly into the intersection of our selected languages' features. This makes it a natural candidate for being the foundation of an interoperable language. Using a subset of an existing language has the practical benefit of eliminating the need for developers to learn a new language.

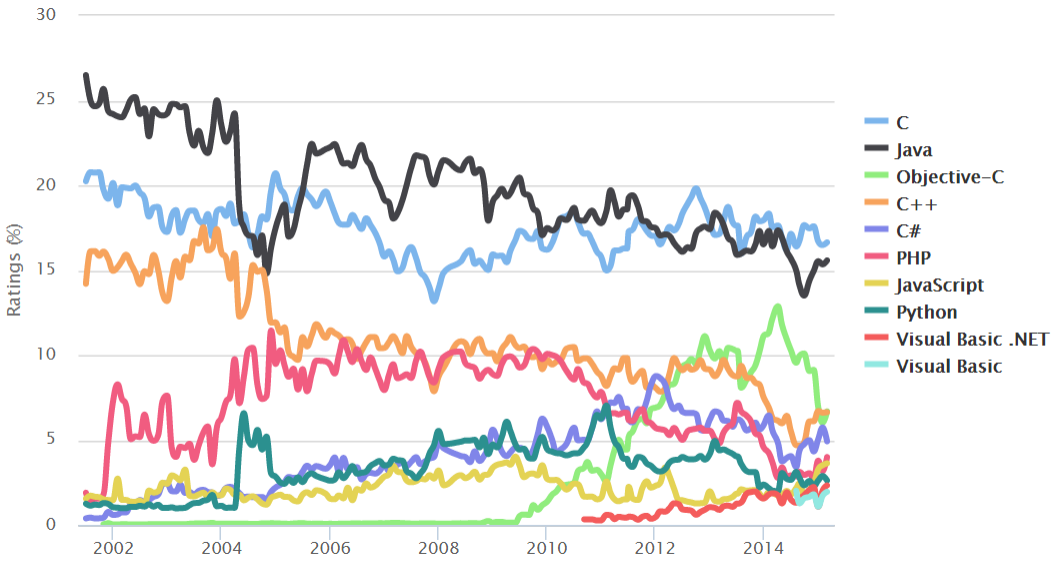

Figure 6.1: TIOBE Index History [48]

For our interoperable component language, we define a subset of Java dubbed *iJava*: interoperable Java. A detailed language specification is available for Java, and its syntax and semantics are familiar to a large portion of the community. A formal language specification for iJava is given in the appendices of this thesis. Here, we define iJava in terms of how it departs from Java.

The following subsections discuss iJava's data types, expressions and operations, statements, subroutines, organizational structures, exception processing, and standard libraries.

6.3.1 Data Types and Variables

In this section, we define iJava's features related to data types and variable definitions, and discuss related concerns. Table 6.1 presents a summary of the relevant language features in each of our surveyed languages, and shows which we chose to include in iJava.

Static vs. Dynamic Typing: As discussed in Section 6.1, iJava uses static typing.

Boolean type: Like all of our surveyed languages, iJava includes a special-purpose boolean type.

Numeric types: Our language analysis identified a common subset of numeric types including signed integers with storage sizes up to 64 bits, and a double-precision floating point type. iJava adopts Java's byte, short, int, long, and double types. We arrived at this subset via the following

observations.

Most of our surveyed languages provide a variety of both integer and floating point types; JavaScript is notable among our set for having only a floating point type. Java, C++, and Object Pascal all provide a collection of integer types with storage requirements ranging from 8 to 64 bits, while Python provides a single integer type that provides unbounded precision. In addition to the variations on integer type sizes, some languages, including Object Pascal and C++, provide types for unsigned integers; Java and Python do not. Current versions of all of our surveyed languages except JavaScript provide an integer data type that meets the range requirements for a 64-bit signed integer.

Among the floating point numeric types, there is less variation: of the languages in our survey, many provide options for single-precision and double-precision floating point values, as in the float and double data types in Java and C++. Java and JavaScript specify IEEE 754 64-bit compatibility for their double-precision type; C++, Python, and Object Pascal leave the precise floating point size and standard as implementation defined. Because of the ubiquity of IEEE 754 and its 64 bit support on modern platforms, we adopt that specification for iJava. However, we note that the degree of reliability that our translator can achieve in the translation of programs that make use of floating point data will rely on the underlying numeric support in the target language implementations.

Other variations of numeric types, such as Object Pascal's fixed-point currency type and Python's complex number type, are excluded from our analysis because they are not supported by all of our languages.

A pure intersection of the numeric data types in our language set would include only a double-precision floating point type. For efficiency, we also include a set of signed integer types ranging from 8 to 64 bits. All of these types except the 64-bit type can be completely, if inefficiently, represented in JavaScript using its double precision numeric type. The 64-bit integer type can be represented in JavaScript using available libraries.

Character and String types: iJava adopts Java's char and String types, following the logic discussed in Section 6.1.

Some of the surveyed languages provide more than one data type for single characters in order to handle more than one type of character encoding. For example, Object Pascal provides Unicode support via its WideChar type, and ANSI encoding via AnsiChar. All surveyed languages

support 16-bit Unicode characters encoded using UTF-16, so that is the character encoding used in our feature subset. We note in passing that the current Unicode standard requires up to 21 bits for encoding some characters; handling characters that cannot be represented with a single 16-bit Unicode code point is an area of future work.

All of the surveyed languages have a data type that can be used to manipulate variable-length character sequences. However, the languages differ on whether the type is a primitive or is constructed from lower-level primitives. For example, Object Pascal, Python and JavaScript provide a primitive string type; Java and C++ provide a string class in their standard library that implements character sequences using their primitive character type.

Java defines the maximum length a string value may have; the others define it as either unbounded, or leave the maximum length as an implementation detail. Thus, our intersection must leave the maximum string length undefined. This is an undesirable state, because it jeopardizes the reliable translation of programs using strings. However, we note that the same reliability issue arises in porting programs between two implementations of the same language (for all languages but Java). In practice, we observe that common implementations of all of our languages on current desktop platforms support strings containing hundreds of thousands of characters, which is sufficient for the reliable translation of many programs.

Array types: iJava adopts Java's array functionality, including both single- and multi-dimension structures. This represents the results of our language intersection analysis that evaluated the diverse characteristics exhibited by the expandable and flexible Python list and JavaScript array, as well as the traditional statically typed arrays of Java, C++, and Object Pascal. Our analysis considered the following factors:

- In statically typed languages such as Java and C++, arrays usually have a fixed element type and capacity; in dynamically typed languages, including JavaScript and Python, the element type can vary, and the capacity can be altered. C++ and Object Pascal allow both statically allocated and dynamically allocated arrays; statically allocated arrays cannot be resized, but dynamically allocated arrays are accessed via reference-type variables, and can be reassigned after instantiation, yielding some resizing capability. A common subset that encompasses all of these characteristics would include dynamically allocated arrays with a fixed element type, accessed via a reference-type variable.

- In Object Pascal, the lower and upper bounds of the arrays are both defined by the programmer, and may be any ordinal type, while in C++, Java, and Python, indexes must be integers, with the lower bound defined as 0. JavaScript treats arrays as a subtype of object values, allowing array indices to be strings as well as integers. The common subset, then, permits array indexing with integers only, with a lower bound of 0.

- In all of our surveyed languages, dynamically allocated multidimension arrays are arrays of dynamically allocated arrays. The instantiation of each dimension involves dynamically allocating subarrays for each slot in the dimension, which need not be of the same length as the other subarrays in the dimension. iJava takes this approach.

We note that iJava's array semantics can be efficiently implemented in all of our surveyed languages.

Record Types: In statically typed languages, including Java and C++, a record type statically specifies the names and types of its member fields. In dynamically typed languages, including Python and JavaScript, record types are generally encoded using a name-value map, in which the member fields are dynamically, rather than statically, defined. A common subset compatible with both approaches, then, would utilize a record type declaration with statically typed member fields.

Function Types: All of the analyzed languages allow functions (or pointers to functions) to be stored in variables and passed to and from subroutines. However, Java does not provide a special purpose function type, instead using interfaces to achieve an effective superset of this capability. Because Java provides no special purpose function type, we choose not to include one in iJava.

Reference and value types: As in Java, iJava's primitive types are value types, while its class and array types are reference types. We arrived at this semantic via the following analysis. We observe that Python and JavaScript variables hold references to values. In Java, variables of primitive type hold values, while array and class-type variables hold references to objects. In C++ and Object Pascal, any variable can be defined to hold either a value or a reference to a value, since all data types are value types, but the programmer may define a variable to hold a pointer to a value. Java's model represents a good subset of our languages: it is compatible with Python and JavaScript, since the numeric and boolean values in those languages are immutable. It is also compatible with translation to C++ and Object Pascal, as it represents a subset of the semantic

options available in those languages.

Memory Management: Related to the issue of reference and value types is the question of memory management. All of our survey languages except C++ can track references to allocated memory and automatically deallocate values that become inaccessible. Since garbage collection libraries are available for C++, we choose to include garbage collection in our subset of common features.

Variable declaration: Python has no variable declaration statement, and JavaScript's `var` statement is optional in some cases. However, a variable declaration statement is required by all statically typed languages and must therefore be included in our language subset. We observe that its inclusion in iJava is compatible with translation to JavaScript and Python, since it can be removed or replaced with an assignment statement. Of our statically typed languages, only C++ allows implicitly typed variable declarations; Java and Object Pascal both require the variable type to be specified in the declaration, so iJava requires explicitly typed variable declarations.

Constants: All of our statically typed languages provide mechanisms to define named constants in the context of a class definition. C++ and Java allow both static and instance constant members, while Object Pascal allows only static constants. Even though JavaScript and Python do not provide mechanisms to define constants, since this is an important feature in static languages, and constants can be emulated in languages that do not support them using variables and naming conventions, we include static class constants in iJava.

Enumerated types: All of our statically typed languages provide a mechanism to define an enumerated data type, while our dynamically typed languages do not provide such a mechanism. We exclude enumerated types from iJava.

6.3.2 Operators and Expressions

In this section, we explore issues involving operators and expression evaluation.

Numeric operations: As shown in Table 6.2, iJava's numeric operations include the standard addition, subtraction, multiplication, division, and modulus operators; numeric cast conversion operators; bit shift; and comparison operators, all using standard Java semantics. This represents the broad consensus exhibited by our language set in this area.

Java specifies the behavior of arithmetic operations that result in numeric overflow and underflow, but iJava leaves these behaviors undefined, since there is variation among the surveyed

	C++	Java	Object Pascal	Python	JavaScript	iJava
Typing						
Static vs. Dynamic	Static	Static	Static	Dynamic	Dynamic	Static
Explicit vs. Implicit	Explicit, Implicit	Explicit	Explicit	N/A	N/A	Explicit
Numeric Types						
Integer Precision	8–64 bits	8–64 bits	8–64 bits	arbitrary	N/A	8–64 bits
Unsigned Ints	Y	N	Y	N	N	N
Floating Point	single, double	single, double	single, double	double	double	double
Character Type	Y	Y	Y	N	N	Y
String Type						
Longest String	undefined	$2^{31} - 1$	unbounded	undefined	undefined	undefined
Mutable?	Y	N	Y	N	N	N
Boolean Type	Y	Y	Y	Y	Y	Y
Arrays						
Memory Alloc	Static, Dynamic	Dynamic	Static, Dynamic	Dynamic	Dynamic	Dynamic
Lower Bound	Implicit	Implicit	Explicit	Implicit	Implicit	Implicit
Records	Y	Y (class)	Y	N	N	Y
Reference / Value Types						
Primitives	Ref, Value	Value	Ref, Value	Value	Value	Value
Arrays	Ref	Ref	Ref, Value	Ref	Ref	Ref
Classes	Ref, Value	Ref	Ref, Value	Ref	Ref	Ref
Function Types	Y	N	Y	Y	Y	N
Garbage Collection	Library	Y	Y (COM based)	Y	Y	Y
Class Constants	Static, Instance	Static, Instance	Static	N	N	Static
Enumerations	Y	Y	Y	N	N	N

Table 6.1: Data Type Analysis

languages in how these errors are handled. In the case of floating-point division by zero, where there is also disagreement among our surveyed languages, we choose Java semantics for iJava, because its behavior can be easily emulated in languages which handle this situation differently from Java.

Logical operations: All of our surveyed languages except Object Pascal provide both logical and bitwise logical operators. Object Pascal provides only bitwise logical operators, but when applied to boolean types, they act as logical operators, so we include them in our analysis as logical operators. C and C++ implicitly convert integer operands of logical operators to boolean values, but Java allows only boolean values for its logical operators. All languages perform conditional evaluation by default on logical operations. Thus, iJava includes both bitwise and short-circuiting logical operators.

Bit shift operations: All of our surveyed languages provide left and right bit shift operators that operate on integers. Java, C++, and JavaScript provide both arithmetic and logical variants of the right shift operator, while Python provides only an arithmetic right shift, and Object Pascal provides only a logical right shift. Although the effects of these operators is well-defined on unsigned integers, their effect on signed integers depends on the underlying numeric representation. C and C++ leave the representation of integers as an implementation detail, and thus certain applications of the left and right shift operators to signed numbers are undefined in the language standard. Java and Python specify two's complement representation, and completely define the behavior of the operators. Because two's complement is a common representation used in C and C++, we choose to include the bit shift operators in iJava, since we can fill in missing operations using library routines. We note that iJava programs that utilize these operators will encounter reliability issues when translated to C and C++ on systems using representations other than two's complement.

String operations: iJava's string operations are provided by the java.lang.String class in its standard library, which includes a subset of the Java String methods, as described in Section 6.3.8. Also, Java's overloaded use of the + operator to perform string concatenation is supported in iJava. Although not all of our languages support the overloaded use of the + operator in this way, the behavior can be easily mapped in translation to target languages.

Pointer expressions: C++ and Object Pascal permit the use of pointer arithmetic to allow a program to address arbitrary memory locations, provide an operation to take the address of arbitrary variables, and allow conversions between pointers and integers. Java, Python, and JavaScript do not provide such operations. Thus, the only pointer-related operations in iJava include

the dereference operator, and equality/inequality comparison. We note in passing that because the result of dereferencing a null reference is undefined in C++, the result must also be undefined in iJava—a departure from Java behavior.

Array expressions: We consider two array operations supported by all of our languages: length determination and indexing. All of our languages except C++ provide an operation to obtain the length of a dynamically constructed array. Since C++ provides array-like containers in its standard library that do provide this operation, we include this capability in iJava.

Array index bounds are checked in Java, JavaScript, Python, and optionally, in Object Pascal (using a compiler switch). Out-of-bounds accesses either trigger a runtime exception, or in the case of JavaScript, they evaluate to the value undefined. C++ does not perform bounds checking on its native arrays, but its standard library includes array-like containers that provide this functionality. Therefore, iJava implements bounds checking.

Expression evaluation order: A common source of difficulty in program porting efforts is the issue of expression evaluation order. Subexpressions that cause side effects can affect the evaluation of other parts of the same expression. If the order of evaluation of expressions is undefined, programs containing expressions with side effects cannot be reliably ported to other language implementations. Python evaluates expressions left-to-right, as does Java and JavaScript. However, C++ and Object Pascal do not define the order of expression evaluation. We leave the order of expression evaluation unspecified in iJava, a departure from Java semantics.

Expressions with side-effects: Java, C++, and JavaScript include several operators, such as the assignment operator, which can be used to modify the values of variables used in expressions. Python and Object Pascal do not include such capabilities, so iJava does not allow the use of such operations in expressions.

Table 6.2 summarizes the expressions and operations discussed in this section.

6.3.3 Statements

In this section, we discuss the statements found in iJava.

Assignment: Like all of our surveyed languages, iJava includes an assignment operator. Java's assignment compatibility rules are the most restrictive of our languages, and we therefore adopt them for iJava. Further, Java requires that local variables be explicitly initialized before first use, so iJava enforces this requirement.

	C++	Java	Object Pascal	Python	JavaScript	iJava
Operators						
Arithmetic	$+ - \times \div$ mod	$+ - \times \div$ mod	$+ - \times \div$ mod	$+ - \times \div$ mod x^y	$+ - \times \div$ mod	$+ - \times \div$ mod
Logical	$\wedge \vee \neg$	$\wedge \vee \neg$	$\wedge \vee \neg$	$\wedge \vee \neg \oplus$	$\wedge \vee \neg$	$\wedge \vee \neg$
Bitwise Logical	$\neg \wedge \vee \oplus$	same	same	same	same	same
Bit shift	logical, arithmetic	logical, arithmetic	logical	arithmetic	logical, arithmetic	logical, arithmetic
Relational	$= \neq > \geq < \leq$	same	same	same	same	same
Evaluation Order	Undefined	Left-Right	Undefined	Left-Right	Left-Right	Undefined
Array Operations						
Length check	Y	Y	Y	Y	Y	Y
Bounds check	Y	Y	Y	Y	Y	Y
Pointer Operations						
Dereference	Y	Y	Y	Y	Y	Y
Compare	Y	Y	Y	Y	Y	Y
Address-of	Y	N	Y	N	N	N
Arithmetic	Y	N	Y	N	N	N

Table 6.2: Operator and Expression Analysis

All languages except Object Pascal support a form of the assignment operator called *augmented assignment,* which combines a binary operator with assignment (ex. `a += b`). Although Object Pascal does not support augmented assignment, since it can be mapped to Object Pascal in a straightforward way, we support it in iJava.

Control statements: iJava includes if/else and while constructs, the only selection and looping statements found universally in our language set (see Table 6.3). Like Java, iJava requires that if conditions and while conditions be boolean-typed.

All of our surveyed languages also include simple, unlabeled break and continue statements used to terminate loops or skip portions of loop iterations, so iJava includes these as well.

Unreachable statements: Java requires that all statements in a method must be reachable. If the Java compiler determines, using static analysis, that there is not some execution path from the beginning of a method to a given statement, then there is a compile error. For example, statements that immediately following a return or a throw statement are unreachable. Our interoperable language must therefore include this constraint.

See Table 6.3 for a summary of the analysis in this section.

	C++	Java	Object Pascal	Python	JavaScript	iJava
Augmented Assignment	Y	Y	N	Y	Y	Y
Branch Statements						
if	Y	Y	Y	Y	Y	Y
switch	Y	Y	N	Y	N	N
break/continue	basic	labeled	basic	basic	labeled	basic
goto	Y	N	Y	N	N	N
Loop Statements						
pre-test (while)	Y	Y	Y	Y	Y	Y
post-test	Y	Y	Y	N	Y	N
for (counting)	N	N	Y	N	N	N
for (C style)	Y	Y	N	N	Y	N
for (enumerating)	Y	Y	Y	Y	partial	N

Table 6.3: Statement Analysis

6.3.4 Subroutines

In this section, we consider subroutine-related issues including recursion, nested subroutines, scoping, parameter passing modes, and method overloading.

Recursion: iJava supports recursion, as do all of our languages. The depth of recursion permitted is generally left as an implementation detail, so iJava does not attempt to define it. We note that the lack of definition of this type of limit has an impact on the reliable translation of programs that is similar to the impact of other undefined limits such as the maximum length of a string, or the maximum amount of memory allocation permitted.

Scoping: Object Pascal, Python, and JavaScript support nested subroutines, but C++ and Java do not (although they do support classes nested inside subroutines), so nested subroutines are excluded from our subset. Statically allocated and initialized local variables, a feature of C++, are not supported in the other languages, and are thus excluded. All of the surveyed languages utilize ALGOL-style static scoping, where every occurrence of a variable is statically associated with its definition by lexical analysis of the enclosing block structure, rather than being resolved at runtime through dynamic lookup in the runtime stack.

	C++	Java	Object Pascal	Python	JavaScript	iJava
Nested Subroutines	N	N	Y	Y	Y	N
Global Subs/Variables	Y	N	Y	Y	Y	N
Scope Resolution	static	static	static	static	static	static
Parameter Modes	value, ref	value	value, ref	value	value	value
Array Parameters	ref	ref	ref (dynamic arrays)	ref	ref	ref
Method Overloading	Y	Y	Y	N	N	Y
Recursion	Y	Y	Y	Y	Y	Y

Table 6.4: Subroutine Feature Analysis

Python, JavaScript, Object Pascal, and C++ permit globally scoped top-level variables and subroutines that are defined outside a class. Java requires all variables and subroutines to be encapsulated inside a class, so iJava does not support top-level variables and subroutines. We note that global access to variables and subroutines can be achieved in iJava through the use of public static declarations in classes.

Parameter passing modes: All of the languages support some combination of by-value and by-reference parameter passing. Java, JavaScript, and Python support only by-value parameters, but regularly achieve a form of by-reference semantics by passing references as values. iJava adopts the model employed by Java and JavaScript, in which primitives are passed by value, and values such as objects and arrays are passed by reference. This represents a reasonable intersection of the parameter passing capabilities of our language set.

Method overloading: Method overloading, a feature allowing several subroutines to use the same name, being distinguished by the number and types of their parameters, is supported in Java, Object Pascal, and C++, but not JavaScript or Python. Since method overloading can be emulated in a straightforward manner using name mangling in the latter languages, we choose to include it in iJava.

Table 6.4 summarizes our discussion of subroutine capabilities.

6.3.5 Object Oriented Features

All of our analyzed languages include object-oriented features, including encapsulation, inheritance, and dynamic binding. In this section, we analyze specific features in this category.

Class definition: All of our languages except JavaScript utilize a class-based approach for defining objects. In contrast, JavaScript uses a prototype approach. Since a class-based approach is inherently more restrictive than a prototype-based approach, our interoperable intersection as embodied in iJava utilizes a static, class-based approach.

Class initialization: All of our languages provide a constructor mechanism to define a block of code that executes when a class is instantiated, to initialize instance variables. iJava also supports the initialization of instance variables at the point of declaration, since this feature can be emulated using constructors in languages such as Object Pascal that do not support this feature. iJava allows the initialization of static class variables at the point of declaration, since all languages include features that can support this.

Inheritance: All of our languages support an inheritance mechanism that supports method overriding and dynamic binding. Java, JavaScript, and Object Pascal permit a class to have at most one immediate ancestor (single inheritance), while Python and C++ allow a class to have multiple immediate ancestors (multiple inheritance). Because single inheritance is more restrictive than multiple inheritance, iJava is restricted to single inheritance.

C++ and Object Pascal provide a way to define both virtual and non-virtual methods; virtual methods support overriding and dynamic binding, while non-virtual methods do not. Java, Python, and JavaScript support only virtual methods; therefore, our iJava includes only virtual methods.

Java, JavaScript, Object Pascal, and Python all utilize a unified class hierarchy, in which there exists an ultimate ancestor base class from which all other classes ultimately derive. C++ does not require a single base class, thus allowing multiple class hierarchies. In our intersection, we restrict iJava to a single class hierarchy with a common ancestor.

All of our surveyed statically typed languages have operators to perform checked downcasts. C++ and Object Pascal also provide operators to perform unsafe unchecked downcasts, but Java does not, so we exclude an unchecked downcast operation from iJava. Dynamically typed languages, by definition, do not require a downcast operator, but its behavior can be supplied via a library function.

C++ and Java both support covariant return types for overriding methods, but require invariant parameter types. Object Pascal, in addition to requiring invariant parameter types, also requires invariant return types. iJava therefore requires invariant return types as well as invariant parameter types for overriding methods.

Abstract Classes and Interfaces: iJava supports both abstract classes and interfaces,

which are important mechanisms provided in statically typed object-oriented languages to enable modular development. Java, Object Pascal, and C++ all provide mechanisms to omit method implementations from abstract classes. Java and Object Pascal also provide a specialized form of abstract class called an *interface* that contains only abstract methods. Both Java and Object Pascal allow interfaces to form an inheritance hierarchy, and for classes to implement multiple interfaces, thereby obtaining some of the benefits of multiple inheritance. C++, which supports multiple inheritance, needs no special interface mechanism, since it can encode interfaces using pure abstract classes. These features are not provided (or needed) in Python or JavaScript, but are compatible with translation to those languages via removal and/or library support, so we include them in our intersection.

Information hiding: C++, Object Pascal, and Java provide enforced information hiding mechanisms for class members, using visibility modifier keywords. All three languages support the public, protected, and private keywords, with the usual semantics. Java and Object Pascal allow classes within the same namespace to freely access each others' members, in effect bypassing the visibility modifiers; in a similar vein, C++ provides a friend mechanism to bypass the access controls. For a clean, simple intersection, we adopt the C++ visibility modifier keywords public, protected, and private, and their C++ semantics, for iJava's information hiding mechanism.

Scripting languages typically do not provide an enforced information hiding feature: JavaScript lacks it entirely, and Python provides naming conventions for information hiding which can be circumvented. Since information hiding is an important language feature that can be viewed as enforcing a restriction on the set of legal programs, we consider it to be in the intersection of our language set, and support information hiding in iJava.

Table 6.5 summarizes our discussion of object-oriented features.

6.3.6 Exception Processing

All of our surveyed languages provide an exception mechanism to report and handle errors. In this section, we discuss the exception processing features of the languages in our set.

All of the languages use the termination model, rather than the resumption model, for exception processing: when an exception is raised, further processing in the method is terminated and a search is made for an exception handler.

Java classifies exceptions into two categories: checked exceptions thrown by a method must

	C++	Java	Object Pascal	Python	JavaScript	iJava
Class vs Proto	Class	Class	Class	Class	Proto	Class
Inheritance	Multiple	Single	Single	Multiple	Single	Single
Interfaces	N	Y	Y	N/A	N/A	Y
Info Hiding	Y	Y	Y	Partial	N	Y
Class Hierarchy	Multiple	Single	Single	Single	Single	Single
Downcast	Safe, Unsafe	Safe	Safe, Unsafe	N/A	N/A	Safe
Methods	Virtual, Nonvirtual	Virtual	Virtual, Nonvirtual	Virtual	Virtual	Virtual
Abstract Methods	Y	Y	Y	N/A	N/A	Y

Table 6.5: Object Oriented Feature Analysis

be explicitly handled by the caller, either by a try statement or by an advertised exception propagation using a throws clause in the method interface; unchecked exceptions do not need to be explicitly handled. The other languages do not include checked exceptions, and iJava therefore includes only unchecked exceptions.

Java and Python allow only instances of a class in an exception class hierarchy to be thrown as exceptions; C++ and JavaScript allow values of any type to be thrown; Object Pascal allows any object type to be thrown. iJava thus restricts throwable values to instances of a class in an exception class hierarchy.

C++ and Java allow a subroutine to advertise the types of exceptions it may throw via an exception specification clause (although this feature is deprecated in recent versions of the C++ standard); the other languages do not. We therefore exclude this feature from iJava.

All of our analyzed languages include a statement to raise an exception (ex. throw in Java), as well as a statement to handle exceptions (ex. try-catch in Java). JavaScript allows only one catch block in its try-catch, while the other languages allow multiple catch blocks to provide exception-specific handling. Since multiple catch blocks are useful and can be easily emulated in JavaScript with a sequence of if-else tests that check the type of exception that was thrown, iJava allows multiple catch blocks. All of the languages that support multiple catch blocks have identical semantics regarding the resolution of matches for a given exception: the first matching catch block applies. Finally, all languages allow an exception parameter in a catch block to receive the specific exception instance that was thrown, so that it can be inspected.

All of our languages except C++ allow the try-catch construct to contain a finalization block,

	C++	Java	Object Pascal	Python	JavaScript	iJava
Checked	N	Y	N	N	N	N
Throwable	Anything	Exception only	Any Object	Exception only	Anything	Exception only
Exception Specs	Y (deprecated)	Y	N	N	N	N
Multiple Catch	Y	Y	Y	Y	N	Y
Finally	N	Y	Y	Y	Y	N

Table 6.6: Exception Feature Analysis

which executes whether or not an exception is raised. Because C++ omits this feature, we exclude it from iJava.

Table 6.6 summarizes the issues surrounding exceptions.

6.3.7 Other Language Features

In this section, we discuss additional language features that we considered in our analysis.

Generics: The languages in our set take a variety of approaches to generic algorithms and data structures. C++, Java, and Object Pascal provide a template facility designed to facilitate generic programming, although the implementation details and capabilities differ considerably. Python and JavaScript provide generic functionality by definition, via their status as dynamically typed languages.

Object Pascal, Java, and C++ all allow classes to be defined using one or more generic type parameters, which can be used to specify types for both member variables as well as parameter and return types for member operations. Java allows constraints on type parameters to be specified, but Object Pascal and C++ do not provide a constraint mechanism. Java and C++ allow subroutines to define generic type parameters, but Object Pascal does not. Therefore, iJava allows generic classes only, with no constraints on generic parameters.

Namespaces: Many languages have structural features to prevent naming conflicts and to group code into hierarchically structured logical units. All of the surveyed languages except JavaScript provide a namespace capability. Since namespaces are regularly implemented in JavaScript programs using a variety of closure-based modular coding patterns, we support namespaces in iJava.

Introspection: The dynamic inspection of data structures is supported to varying degrees in all of our surveyed languages. All of the languages provide a mechanism to query the type of an object to determine whether it is a subclass of a specified type; examples of this operator include

	C++	Java	Object Pascal	Python	JavaScript	iJava
Generics	Y	Y	Classes only	N/A	N/A	Classes only
Parameter Constraints	N	Y	N	N/A	N/A	N
Exception Specs	Y (deprecated)	Y	N	N	N	N
Instance Of Operator	Y	Y	Y	Y	Y	Y
Namespaces	Y	Y	Y	Y	N	N
Operator Overload	Y	N	Y	Y	N	N
Threads	Y	Y	Y	Y	N	N

Table 6.7: Miscellaneous Feature Analysis

dynamic_cast in C++, instanceof in Java, and isinstance() in Python. Therefore, we include that operation in our subset. Other introspection features, such as dynamically discovering the fields and methods available in a given object, are not available in C++ and Object Pascal, so we exclude such functionality from iJava.

Operator overloading: Operator overloading is supported in Object Pascal, Python and C++, but not in Java or JavaScript. It is thus excluded from iJava.

Threads: All of the analyzed languages except JavaScript provide capabilities for starting and manipulating independent threads. Because JavaScript lacks support, we exclude threads from iJava.

Table 6.7 summarizes the discussion in this section.

6.3.8 Standard Library

The iJava standard library includes a minimal subset of classes from the Java standard library that are needed for basic language support. It includes partial implementations of the following (details are provided in Appendix A):

- The Object and String classes
- The Exception, Error and RuntimeException classes, along with subclasses used to report certain errors
- The wrapper classes Boolean, Character, Byte, Short, Integer, Long, and Double, which are needed to wrap primitives into objects

Chapter 7

Translation of Language Features

In this chapter, we discuss details concerning the translation of Java to Python and C++ using our translation system introduced in Chapter 5. We begin in Section 7.1 with a discussion of the mapping of Java features to iJava. Then, Section 7.2 presents the translation of iJava features to Python and C++.

7.1 Mapping Java to iJava

As we discuss in Chapter 8, we evaluate our system using Java test cases. These test cases use Java features that are excluded from iJava as the result of design choices imposed by our methodology described in Section 6.1. Our translation system transforms these constructs to iJava features before translating them to Python or C++. In this section, we discuss the mapping of these Java features to iJava structures. It is not our purpose to establish a complete mapping from Java to iJava; since iJava, by design, omits key primitives such as synchronization and support for multithreading, such a mapping is not possible.

7.1.1 Pre/Post Increment Expressions

iJava excludes expressions with side effects, for example the pre- and post- increment and decrement operators (e.g., `++x`, `--x`, `x++`, `x--`). In cases where they occur in a statement by themselves, our translator transforms these operators to augmented assignment operations (e.g., `x++` is transformed to `x += 1`). Our test cases did not make use of these operators in compound

expressions.

7.1.2 Loops

iJava provides a while loop, but not Java's for, do, or enumerating for loop. Since our test cases made use of these loops, we transformed each of these into while loops. For example, consider a do-while loop having the structure shown in Listing 7.1. The translator transforms it to a iJava while loop, adding a locally defined flag variable unique to the loop (named here firstTime) that ensures that the loop body will execute at least once. iJava's short-circuiting rules ensure that the loop condition is not evaluated upon initial entry into the loop, but rather is evaluated at the end of each loop iteration that terminates normally. This transformation thus preserves the semantics of Java's do-while loop.

Listing 7.1: Java Do-while example

```
do {
   loop-body
} while ( condition );
```

Listing 7.2: iJava Do-while translation

```
1 boolean firstTime = true;
2 while (firstTime || condition) {
3    firstTime = false;
4    loop-body
5 }
```

7.1.3 Nested and Anonymous Classes

Our test cases contained nested and anonymous classes. Our translator converted these to standalone classes in iJava, handling the scoping issues that occur when the code in the inner class referenced members in the outer class, as shown in Listing 7.3, where *innermethod()* accesses c, a member of Outer. Listing 7.4 demonstrates the transformation, in which the Inner class becomes a top-level class, Outer_Inner, which holds a reference to Outer, outer. This reference enables the refactored innermethod() to access c in Outer.

Listing 7.3: Java inner class example

```
class Outer {

  int c;

  public void useInner() {
    Inner cl =
      new Inner();
    cl.innermethod();
  }

  class Inner {
    public void innermethod() {
      c = 5;
    }
  }

}
```

Listing 7.4: iJava inner class translation

```
1  class Outer {
2
3    int c;
4
5    public void useInner() {
6      Outer_Inner cl =
7        new Outer_Inner(this);
8      cl.innermethod();
9    }
10 }
11
12 class Outer_Inner {
13
14   Outer outer;
15
16   public Inner(Outer outer) {
17     this.outer = outer;
18   }
19
20   public void innermethod() {
21     this.outer.c = 5;
22   }
23 }
```

7.2 Mapping iJava to Python and C++

In this section, we discuss the mapping of iJava features to Python and C++.

7.2.1 Names and Namespaces

Field and variable names in iJava are mapped unchanged to C++ and Python, except where the identifier is a keyword in the target language. These situations are resolved by appending

symbols to the identifier.

Although Python and C++ provide a namespace mechanism that performs the same role as the iJava package mechanism, we found it expedient to translate iJava packages by applying name mangling to class and interface names. Class and interface names are mapped to C++ and Python by having their package name prepended to the type name. For example, an iJava type named `foo.bar.Item` would be mapped to a Python or C++ class named `foo_bar_Item`.

7.2.2 Classes and Interfaces

The translator maps each iJava class and interface to a class in the target language. Each member method and instance variable is mapped to a corresponding method and variable in the target class. This mapping is straightforward for C++; for Python, which does not support method overloading, name mangling is used to implement method overloading. Additional work is required to support the mapping of overloaded constructors to Python: since a Python class may contain only a single constructor, we map iJava constructors to static factory methods in the target Python class. For an example of this mapping to static factory methods, see Line 9 of Listing 7.9, which shows the instantiation of an exception in a fragment of the Apache Math Commons library, and its translation on Line 12 of Listing 7.10, which invokes the factory method constructor_R to instantiate the translated exception class.

Information hiding: Our translator does not implement information hiding; this is an item for future work. However, we observe that, given iJava's information hiding semantics, the mapping of the visibility modifiers from iJava to C++ is straightforward. A mapping to Python would involve using Python's naming conventions for protected and private visibility—prefixing member variable names with one or two underscores, respectively.

Interfaces: Neither Python nor C++ have an interface mechanism, so we emulate the feature in both languages. For C++, iJava interfaces map to classes containing only pure virtual methods. Interface implementation is modeled using virtual inheritance. For Python, we utilize the Python Abstract Base Class module, generating classes containing methods defined with the `@abstractmethod` decorator.

Static data initialization: The order of static data initialization between modules is undefined in C++. This can result in incorrect initialization in situations where a static field initialization in one module references a static field in another module, and the second module is initialized after

the first. We addressed the issue of static data initialization for each of our test cases by managing the order in which the modules were linked to control the order in which the modules were initialized. However, an alternative solution would entail wrapping each static class variable as a local static variable in a static C++ class accessor method, and transforming all references to the static variable in C++ to static accessor method invocations.

As an example of how this technique could be applied, consider Listing 7.5. A straightforward translation to C++ yields two modules: one for each class. If the static members in Class1 and Class2 are mapped to regular C++ static member variables, the value of Class2::b would be undefined, since its value would depend on whether its static initialization occurred before or after Class1's initialization. In Listing 7.6, the static members in Class1 and Class2 are wrapped in static methods, and the order of initialization defined in the C++ standard ensures that Class1::a has a well-defined value which matches the expected order.

Listing 7.5: Java static initialization example

```
class Class1 {

  static int a = 5;

}

class Class2 {

  static int b = Class1.a;

}
```

Listing 7.6: C++ mapping

```
1  class Class1 {
2
3    static int& a();
4
5  };
6
7  int& Class1::a() {
8    static int a = 5;
9    return a;
10 }
11
12 class Class2 {
13
14   static int& b();
15
16 };
17
18 int& Class2::b() {
19   static int b = Class1::a();
20
21   return b;
22 }
```

7.2.3 Data Types and Variables

Figure 7.1 shows how iJava's basic types map to C++ and Python. We use a custom support library class to represent strings in both C++ and Python. In both cases, this class implements the Java string methods used in our test cases.

iJava arrays are mapped to C++ using a custom support library wrapper class, which handles the bounds checking. For Python, iJava arrays are modeled as standard Python lists. As an example translation of Java arrays to Python, see Listing 7.9, which presents a fragment of the Apache

iJava	C++	Python
byte	int8_t	int
short	int16_t	int
int	int32_t	int
long	int64_t	int
double	double	float
char	char16_t	str
boolean	bool	bool
String	java_lang_String*	jcl_String

Figure 7.1: Mapping of iJava types to C++ and Python

Commons Math library, and its translation to Python in Listing 7.10. Note how the length check on Lines 2 and 3 of Listing 7.9 maps to Python's len() function on Lines 3 and 5 of Listing 7.10.

7.2.4 Expressions and Statements

Expressions in iJava map in a straightforward fashion to C++ and Python. Some operators are implemented using library functions, in order to achieve semantic parity. For example, Line 3 of Listing 7.7 shows a dynamic cast that occurs in a fragment of the PureMVC library. In the translation to C++ shown in Listing 7.8, the dynamic cast has been mapped to an invocation of the jcl_dynamicCast library function in Line 5. All iJava statements map cleanly to their counterparts in C++ and Python.

Listing 7.7: Java dynamic cast example

```
1 for( int i=0; i<observers.length; i++ )
2 {
3     IObserver observer = (IObserver)observers[i];
4     observer.notifyObserver(note);
5 }
```

Listing 7.8: C++ cast translation

```
1  int32_t i_= 0;
2  while ((i_ < (observers_)->length()))
3    {
4      org_puremvc_java_interfaces_IObserver* observer_=
5          jcl_dynamicCast<org_puremvc_java_interfaces_IObserver*>(
6              observers_->AT(i_));
7      (observer_)->notifyObserver(note_);
8      i_ += 1;
9    }
```

7.2.5 Memory Management

The issue of garbage management can be addressed for C++ through the use of a garbage collection library, such as the well-known Boehm-Demers-Weiser library [5]. We chose not to implement it in our system; this is an area for future work.

7.2.6 Generics

We implemented partial support for generic classes that was sufficient to meet the demands of our library test cases. Our translator handles code that instantiates and uses generic standard library classes, but the translation of code that defines generic classes is part of our future work.

iJava generics are implemented in C++ using templates, with a straightforward mapping. For Python, which needs no generic features due to its dynamic type system, generic parameters are simply removed in translation.

7.2.7 Exception Handling

iJava's exception handling constructs map in a straightforward way to Python and C++. In some cases, it is necessary to map a Java exception type to a corresponding type in the target language standard library. For example, Listing 7.9 shows a fragment of code from the Apache Math Commons library that handles a possible out of bounds array access in the catch block on Line 8. In the translation to Python, depicted in Listing 7.10, the translator mapped the ArrayIndexOutOfBoundsException type to Python's IndexError type on Line 11.

Listing 7.9: Java try-catch example

```
try  {
    for (int i = 0; i < selectedRows.length; i++) {
        for (int j = 0; j < selectedColumns.length; j++) {
            subMatrixData[i][j] = data[selectedRows[i]][selectedColumns[j]];
        }
    }
}
catch (ArrayIndexOutOfBoundsException e) {
    throw new MatrixIndexException("matrix dimension mismatch");
}
```

Listing 7.10: Python try-except translation

```
try:
  i = 0
  while (i < len(selectedRows)):
    j = 0
    while (j < len(selectedColumns)):
      subMatrixData[i][j] = self.data[selectedRows[i]][selectedColumns[j]]
      j += 1

    i += 1

except IndexError as e:
  raise org_apache_commons_math_linear_MatrixIndexException.constructor_R(
     jcl_String("matrix dimension mismatch"), None)
```

Chapter 8

Evaluation and Testing

In this chapter, we present an evaluation of our system using two open source Java libraries as test cases for our system. We begin by describing our test environment. In Section 8.2 we introduce the two libraries, or test cases, and in Section 8.3, we describe our approach to evaluating the test cases and preparing them for translation into Python and C++. In Section 8.4 we describe our use of the expected output of the test cases together with *execution traces* to evaluate the reliability of the generated Python and C++ code, and in Section 8.5 we describe our use of software metrics to evaluate the quality of the generated code. Finally, we conclude this section with some results that describe the performance of the generated code with respect to execution speed.

8.1 Test Environment

We performed our tests using a computer running Windows 10 Professional. Generated Python code was executed using CPython 3.4.2. Generated C++ code was compiled with the Visual Studio 2013 C++ compiler using the standard optimizations configured for the Release Configuration.

8.2 Test Cases

In this section, we introduce the two libraries that we used as test cases to evaluate our translation system: PureMVC, a Model-View-Controller (MVC) framework, and Apache Commons Math, a collection of mathematical algorithms. Both libraries include an extensive suite of unit tests

	PureMVC	Apache Commons Math
Lines of Code	1,456	15,566
Modules	51	195
Test Points	100	3,027

Table 8.1: Test Cases

implemented in the JUnit unit testing framework.

8.2.1 PureMVC

The PureMVC framework facilitates writing applications based upon the Model-View-Controller architectural pattern [38]. It implements several design patterns defined by Gamma et al. [18], including Facade, Command, Mediator, Observer, and Proxy. Versions of the framework are available for multiple languages, including Java, C++, and Python. Our tool translated the entirety of version 1.1 of the Java implementation to Python and C++. In addition to its focus on design patterns, PureMVC also makes liberal use of template features.

8.2.2 Apache Commons Math

Apache Commons Math is a Java library of mathematical and statistical algorithms [2]. This test case makes greater use of the Java standard library than PureMVC, invoking APIs such as file I/O, serialization, reflection, cryptography, and text formatting. We implemented many of these APIs in our Python and C++ support libraries to enable the conversion and testing of the majority of version 1.0 of the library in our tests.

8.2.3 Test Case Metrics

Table 8.1 presents metrics about the test cases. *Lines of Code* is defined as nonblank, noncomment lines of code, and includes both library code as well as the accompanying unit test code. The *Modules* count refers to individual Java source files. We use the term *Test Point* to refer to a single assertion in a unit test.

8.3 Test Case Evaluation and Preparation

In this section, we describe the procedure that we used to evaluate and prepare the target applications that we translated using our system. This evaluation included the identification of third party libraries used by the two test cases. These third party library dependencies included Application Programmer Interfaces (APIs) supplied by the Java Standard Library and the JUnit Testing Framework. To facilitate our translation, we implemented some of these third party libraries and included the translations in our Python and C++ runtime support systems.

After implementing the required dependencies, we made minor modifications to the test cases, detailed in the next section. Finally, we used our translation system to translate the modified test case code to C++ and Python.

8.3.1 Modifications to PureMVC

In our initial testing, we made a single modification to the PureMVC code to facilitate the translation to C++. The modification involved marking a class as *implementing an interface*. This omission in the original PureMVC Java implementation code was an acknowledged oversight by the PureMVC author. We submitted a patch with our modification to the PureMVC author, who accepted it into the official PureMVC library. At the time of writing, our translator converts the entire PureMVC library to Python and C++ with no modifications.

8.3.2 Modifications to Aapache Commons Math

In this section, we describe the modifications to the Apache Commons Math test case to facilitate our tests. Figure 8.1 lists the number of modules in Apache Commons Math that our tool translated in whole or in part. Our tool translated 89% of the lines of code, representing a complete or partial translation of 93% of the modules.

The 13 modules that we did not translate relied on Java APIs that we did not port to the Python and C++ runtime systems, including I/O, formatting, and reflection features. An automated translation of these APIs is part of our future work. We removed code from 15 other modules for three reasons:

1. Some code utilized Java APIs that we did not port to the Python and C++ runtime libraries, including facilities such as serialization, cryptography, and I/O.

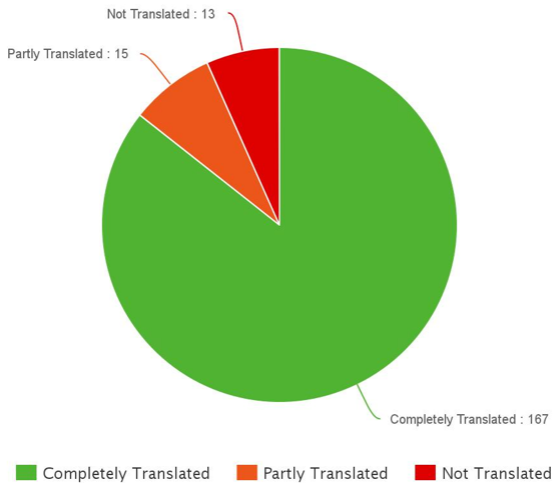

Figure 8.1: Apache Commons Math Classes Translated

2. A portion of the code defined two overloaded methods: one performed a computation on a float-type parameter, and the other performed an identical computation on a double-type parameter. iJava supports the double type but not the float type, and our translator maps the float type to double in its translation process. This resulted in a duplicate method conflict in the translated module. We removed the float overload as superfluous in this case. If the overloads had performed different processing, and removal was not an appropriate action, a more sophisticated overload resolution strategy would have been implemented.

3. Finally, some of the Apache Commons test code attempts to catch Null pointer exceptions. Since the behavior of null pointer dereferences is undefined in iJava, we removed those checks. An alternative approach, requiring additional engineering effort, would involve suppressing these checks automatically during translation, and this effort is part of our future work.

8.4 Reliability Tests and Execution Traces

In this section, we describe the approach we used to evaluate the reliability of the Python and C++ applications automatically generated by our translation system.

The two test cases, PureMVC and Apache Commons Math, both include extensive unit tests written using the widely known JUnit Framework [26]. Our basic approach to evaluating the reliability of our translation is to convert both the libraries and their accompanying unit tests to

the target languages, and then to execute the translated unit tests and compare the results. A key aspect of our analysis involves collecting a *test trace* of the execution of each unit test, which consists of a log of the assertions performed by the unit test as it executes. We compare the test traces of the original unit test executions with those resulting from the translated versions; this comparison allows us to detect certain types of errors that would not trigger a unit test failure in the translated code, providing a more robust reliability analysis than would be afforded by a simple pass/fail test analysis.

Section 8.4.1 describes the procedure we use to capture a trace of execution of the unit tests of the source Java test cases. Section 8.4.2 describes our methodology for executing the translated tests and capturing a test trace. Finally, Section 8.4.3 describes the approach we used to compare and evaluate the test traces.

8.4.1 Execution of Original Unit Tests

To provide a baseline for evaluating the reliability of our translator, we execute the original unit tests that are included in both of our test cases. In addition to noting the individual pass/fail results, we capture a trace of the checks performed during the execution of the unit tests. This trace contains the arguments passed to each JUnit assertion check, generally containing both expected and actual values, and shows the sequence in which the unit tests are executed by the JUnit test framework. Since JUnit has no built-in facility for capturing a trace of its unit tests, we developed a custom tool using BTrace [7], a JVM instrumentation facility, to capture the trace data.

8.4.2 Execution of Translated Unit Tests

After translating the test cases to the target languages, we execute the translated unit tests in order to exercise the translated library code. During the execution of the unit tests, our implementation of the JUnit Testing Framework in the Python and C++ runtime support libraries records a trace of the unit test assertions.

8.4.3 Comparison of Test Traces

After generating the three sets of test traces for both the original library and its translation to C++ and Python, we compare the test traces and analyze any differences. To compare a test trace

from a translated unit test with a test trace from its corresponding original unit test, we developed a tool that filters out differences due to the way our test traces are captured and recorded in JUnit and in the test frameworks in C++ and Python.

In our comparison of the test traces, we compare each test point in the original trace with the corresponding test point in the target trace. This allows us to detect scenarios where the test framework indicates that a given test passed, but the test point values indicate that there was a difference in the expected and/or actual values for one or more assertions within the test.

As an illustration of the data captured in the test traces, Listings 8.1 and 8.2 present a portion of the trace from the Apache Commons Math original unit test and the translated Python unit test, respectively. The traces show the execution of two unit test methods: testSinZero in BisectionSolverTest, and testQuinticZero in BrentSolverTest. The lines that begin with assertEquals show the arguments passed to a given invocation of the JUnit assertEquals method during the unit test, containing the expected value, the actual value computed by the function under test, and the error tolerance allowed for the comparison. The test comparison tool reports a complete match for this fragment, even though lines 6 and 8 have minor differences between the two traces due to the way the number 0 is recorded in the trace file in our JUnit trace capture and the way it is recorded in our Python trace capture file.

8.4.4 Reliability Test Results

In Table 8.2, we summarize the results of our reliability tests. The Java entry shows that the original unit tests included with the source Java code bases all passed. The C++ and Python results show what percentage of the test traces captured from executing the translated C++ and Python unit tests matched the corresponding test points in the test traces captured from executing the source Java unit tests.

Listing 8.1: Trace of JUnit Test

```
1 ...
2 org.apache.commons.math.analysis.BisectionSolverTest.testSinZero
3 assertEquals:null`3.141592502593994E+00`3.141592653589793E+00`1.000000000000000E-06
4 assertEquals:null`3.141592621803284E+00`3.141592653589793E+00`1.000000000000000E-06
5 org.apache.commons.math.analysis.BrentSolverTest.testQuinticZero
6 assertEquals:null`2.775557561562891E-17`0.000000000000000E+00`1.000000000000000E-06
7 assertTrue:null`true
8 assertEquals:null`-7.811562634330656E-11`0.000000000000000E+00`1.000000000000000E-06
9 ...
```

Listing 8.2: Trace of Unit Test (Python Translation)

```
...
org.apache.commons.math.analysis.BisectionSolverTest.testSinZero
assertEquals:null‘3.141592502593994E+00‘3.141592653589793E+00‘1.000000000000000E-06
assertEquals:null‘3.141592621803284E+00‘3.141592653589793E+00‘1.000000000000000E-06
org.apache.commons.math.analysis.BrentSolverTest.testQuinticZero
assertEquals:null‘2.775557561562891E-17‘0‘1.000000000000000E-06
assertTrue:null‘true
assertEquals:null‘-7.811562634330656E-11‘0‘1.000000000000000E-06
...
```

In our PureMVC tests, our analysis of the Python traces showed that all test points matched. For C++, all of the tests that used language features defined in iJava matched. Two of the test points were generated by code that used a language feature not defined in iJava, and as expected, since our C++ compiler handled this feature differently than Java and Python, those did not match.

In our Apache Commons Math tests, our analysis of the test point value differences revealed a variety of causes for failure to match test values in the source Java unit test traces. We noted differences due to the following factors:

- **Undefined behavior:** The translated code invoked behaviors that are undefined in iJava. Examples of this include numeric overflow handling, numeric formatting, parameter evaluation order, and circular dependencies between modules.

- **Standard library facilities:** The implementation of our standard library facilities differed from or omitted features present in the Java standard library.

- **Random number tests**: Some unit tests had behavior which was determined by the generation of random numbers, and thus produced different traces each time the unit tests were run. This meant that portions of the traces from two separate executions of the original Java unit tests did not match each other, and it follows that the corresponding test points in the Python and C++ unit test traces did not match those from the original Java tests. However, these translated unit tests passed their internal assertions, and manual inspection of the traces revealed their behavior was as expected.

	PureMVC Tests Passed	Apache Commons Math Tests Passed
Java	100%	100%
C++	98%	99%
Python	100%	96%

Table 8.2: Reliability Test Results

8.5 Quality Tests

In this section, we discuss the approach we use to validate the quality of the translation generated by our implementation.

8.5.1 Quality Metrics

To evaluate the size and complexity of the translated code, we use two well-established measures: Lines of Code (LOC) and McCabe Cyclomatic Complexity [30]. In order to obtain these measures using a consistent definition, we compute the metrics for the source test cases and the Python and C++ translations using *metrics* [39], a tool that computes metrics for several languages. We validate the computation of the Cyclomatic Complexity metric using a tool we developed [42] using libclang, an interface to the Clang compiler API.

To demonstrate that the generated code preserves the class structure of the original, we compute the number of classes using information produced by Doxygen [11], a documentation tool which supports our source and target languages.

8.5.2 Quality Test Results

Table 8.3 and Table 8.4 present the results of our quality metrics analysis for PureMVC and Apache Math Commons, respectively. Here, we discuss those results.

In both cases, the number of classes increased during the translation to C++ and Python. The difference is due to the handling of anonymous inner classes during translation. The metrics tool did not detect anonymous inner classes in the Java source, and those were not included in the count. During translation, the anonymous inner classes were converted to regular classes in C++ and Python, and were subsequently detected by the metrics tool and included in the counts.

	Classes / Interfaces	McCabe	LOC
Java	52	37	1,456
C++	54	37	2,477
Python	54	83	1,650

Table 8.3: Quality Test Results: PureMVC

	Classes / Interfaces	McCabe	LOC
Java	185	904	13,892
C++	193	904	21,397
Python	193	1,111	13,732

Table 8.4: Quality Test Results: Apache Math Commons

The increase in McCabe complexity for Python in both cases is due to the way our translator handles overloaded constructors for Python, which does not support more than one constructor per class. Our mapping introduced an additional selection statement for each constructor in order to handle the constructor overloading; this increased the complexity metric by 1 for each constructor, which accounts for the increase in the metric. Our Clang-based analysis tool reproduced the results produced by the Python tool.

The LOC metric for Python is very close to the Java count, but a word about the increase in the C++ LOC is in order. C++ programs split the definition of a class between .h and .cpp files, and the syntactic duplication introduced by this organization tends to increase the LOC counts for C++ programs, as compared to languages such as Java in which a class definition is contained in a single file. This duplication accounts for the larger LOC counts we observed.

8.6 Performance Tests

In this section, we discuss the approach we use to evaluate the performance of the code generated by our implementation, and present the results of our evaluation.

	Java	C++	Python
PureMVC	0.8	0.2	1.7
Apache Commons Math			
Linear	2.7	4.2	14.4
Analysis & Complex	3.1	0.6	5.4
Distribution	7.2	10.8	187.0
Statistics	22.9	43.5	475.0
Misc	3.6	0.5	6.2
Total Commons Math	39.5	59.6	688.0

Table 8.5: PureMVC and Apache Performance Test Results. All times are in seconds.

8.7 Performance Test Methodology

We evaluate the performance of our translations by timing the execution of the translated unit tests. Our platform is a Dell Latitude laptop with a 1.7 GHz Intel i3 processor and 8 MB RAM. We time the original Java unit tests, and use those timings as the basis for evaluating the timings of the translated unit tests. This results in a performance comparison between Java, C++, and Python, and we acknowledge in advance the difficulties in drawing conclusions from such a comparison.

In all cases, our timings exclude the initial test program load time, and include only the time to execute the unit tests. We measure wall clock time and average several timings to arrive at each result, excluding outlier timings from the average.

8.7.1 Performance Test Results

Table 8.5 shows timings for the PureMVC and Apache Math Commons test cases. The numbers represent the time required to execute 500 iterations of the unit tests, in seconds. For the Math Commons test case, the times are broken down by module, for further granularity.

The PureMVC results are unsurprising. This is a relatively simple library; its operations involve basic manipulations of lists and maps. The unit tests themselves do not involve large amounts of data, and in such a scenario, we would expect the timings to favor C++, a statically compiled language. The numbers bear that out.

The Apache Math Commons results reveal Java as the clear winner over C++ and Python,

Listing 8.3: Python loop iteration performance test

```
1 k = 0
2 def foo():
3     global k
4     k += 1
5
6 i = 0
7 while i < 1000000000:
8     foo()
9     i += 1
```

	Execution time
Java	4.2s
C++	0.6s
Python	638.2s

Table 8.6: Basic Iteration Performance Test Results. All times are in seconds.

although the module breakdown shows that C++ wins significantly over Java in two of the modules. Some explanation of the Python performance numbers for the Distribution and Statistics modules is in order. We observed that some of the algorithms executed in those test involve loops with a large number of iterations, and devised a test to check the relative performance of Java, C++, and Python for executing simple loop iterations. Listing 8.3 shows our performance test code for Python; we created equivalent versions for C++ and Java. To prevent the C++ compiler from optimizing away the loop, we have the loop body invoke a function that increments a global.

Table 8.6 shows the execution time of our basic loop iteration test for each language implementation. Python's performance relative to C++ and Java clearly reveals a significant disadvantage when an algorithm involves a tight loop with a high number of iterations. We conclude that the slow Python performance in the Distribution and Statistics modules is due to the nature of the algorithms in those modules, rather than inefficiency introduced by our translation methodology.

We argue that, while the performance results for the translations of Apache Commons Math overall are not optimal, they are acceptable, given that our translation efforts favored simplicity and readability over performance.

Chapter 9

Conclusions and Future Directions

In this thesis, we presented an interoperable object-oriented language for defining reusable software components, together with a source-to-source translation system for converting components written in this language to Python and C++. Our goal was to demonstrate that a language designed through an intersection-based analysis of a set of popular object-oriented languages is rich enough to express useful components. As our evaluation and results demonstrate, our system is capable of translating substantial test cases with good reliability and acceptable efficiency.

There are several possibilities for future work, including extending our translator to support additional target languages; handling unimplemented iJava language features; translating Java runtime support library dependencies to target languages; and integrating a garbage collection library for C++. Also, although this work focused on object-oriented languages, similar work remains to be done in the functional language space.

In addition, the results of our language analysis can have benefits in the realm of computer science education. For example, an introductory object-oriented programming curriculum could be designed around the common features identified by our analysis. Such a curriculum would focus on the essential features needed to construct useful programs, and by reducing or eliminating coverage of tangential features, could increase time given to problem solving and other important introductory topics.

Source-to-source translation has long been an underutilized tool in the language interoperability tool chest. We believe that the problems confronting comprehensive source-to-source translators have discouraged research in this area, and trust this work has demonstrated the potential

usefulness of source-to-source technology in the development of reusable software libraries.

Appendices

Appendix A iJava Language Specification

A.1 Introduction

This appendix defines *iJava*, a general-purpose, object-oriented language. It is intended to form a subset of Java, so that legal iJava programs are also legal Java programs. To facilitate comparisons with Java, this specification is organized with sections that correspond closely with those in *The Java Language Specification, Java 8 Edition* [20], and some aspects of this specification reference or are drawn from that document.

This language definition is organized as follows: Section A.2 describes the lexical structure. Sections A.3 and A.4 describe the data types, values, conversions, and issues surrounding variables. Section A.5 describes the organizational structure of iJava programs. The remaining sections describe various iJava language structures and their semantics.

A complete BNF grammar for iJava is provided in Appendix B.

A.2 Lexical Structure

iJava programs are written using sequences of ASCII characters. These sequences comprise elements including white space, comments, and tokens. Tokens are keywords, identifiers, literals, separators, and operators.

Line terminators are sequences of ASCII characters which divide the input source file into lines. Lines are terminated by the ASCII characters LF, CR, or the sequence CR LF.

White space is defined as the ASCII space character, horizontal tab, and form feed characters, as well as line terminators. White space separates tokens.

The *separators* consist of the following ASCII characters: ( ) [ ] ; , .

A.2.1 Comments

iJava allows two kinds of comments:

- Multiline comments begin with the sequence /* and conclude with the sequence */ (as in C and C++).
- Single line comments begin with the sequence // and conclude with the next line terminator encountered.

A.2.2 Identifiers

An *identifier* begins with a letter, and consists of a sequence of one or more of the following symbols: letters, digits, the underscore (_) and dollar sign ($). Two identifiers are the same only if they consist of precisely the same sequence of ASCII characters (ex. Count and count are two separate identifiers).

A.2.3 Keywords

The following keywords are reserved and cannot be used as identifiers:

abstract	boolean	break	byte	catch
char	class	continue	double	else
extends	final	if	implements	import
instanceof	int	interface	long	new
package	private	protected	public	return
short	static	super	this	throw
try	void	while		

The following words are also reserved, even though they are not used as keywords, because they are reserved in Java:

assert	case	const	default	do
enum	finally	float	for	goto
native	strictfp	switch	synchronized	throws
transient	volatile			

A.2.4 Literals

A *literal* is a representation of a null, boolean, numeric, or String value in the source program.

Integer literals may be expressed in base 10 (decimal), base 8 (octal), or base 16 (hexadecimal). Their syntax is specified in Section 3.10.1 of *The Java Language Specification*.

Floating point literals in iJava have the syntax specified in Section 3.10.2 of *The Java Language Specification*, with the exception that only double-type literals, not float-type literals, are valid.

There are two literals that correspond to the values of the boolean type: true and false.

Character literals are enclosed in ASCII single quotes, and may include the escape sequences specified in Section 3.10.6 of the Java Language Specification.

String literals are a string of zero or more characters enclosed in ASCII double quotes.

The null literal represents the null reference.

A.2.5 Operators

The following tokens are the operators:

```
= > < ! ~ ? :
== <= >= != && ||
+ - * / & | ^ % << >> >>>
+= -= *= /= &= |= ^= %= <<= >>= >>>=
```

A.3 Types, Values, and Variables

The types of iJava are divided into the primitive types and reference types. The primitive types include the boolean type and the numeric types (byte, short, int, long, char, and double). The reference types are the class types, interface types, and array types. Values of a reference type are references to objects, where an object is defined to be an instance of a class type, or an array. The special null type has a single value, null.

A.3.1 Primitive Types

The primitive types include the boolean type and the numeric types. The integral types and their ranges are as follows:

- byte: -127 to 128, inclusive
- short: -32,768 to 32,767, inclusive
- int: 2,147,483,648 to 2,147,483,647, inclusive
- long: 9,223,372,036,854,775,808 to 9,223,372,036,854,775,807, inclusive
- char: 0 to 65,535, inclusive

The floating point type is double, representing the double-precision 64-bit format IEEE 754 values and operations specified in *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Standard 754-1985 (IEEE, New York).

The IEEE 754 standard includes not only positive and negative sign-magnitude numbers, but also positive and negative zeros, positive and negative infinities, and a special Not-a-Number (NaN). The NaN value is used to represent the result of certain operations such as dividing zero by zero. A NaN constant is predefined as Double.NaN.

Except for NaN, floating-point values are ordered. In order from smallest to largest, they are negative infinity, negative finite nonzero values, negative zero, positive zero, positive finite nonzero values, and positive infinity.

Any value of a floating-point type may be cast to or from any numeric type. There are no casts between floating-point types and the type boolean.

Numeric operations include:

- Comparison operators, which produce a value of type boolean
- Numerical operators, which produce a value of type int or long
- Conditional operator
- Bitwise logical operators
- Bit shift operators
- Cast operator
- String concatenation operator

If at least one of the operands to a binary operator is of floating-point type, then the operation is a floating-point operation, even if the other is integral.

If at least one of the operands to a numerical operator is of type double, then the operation is carried out using 64-bit floating-point arithmetic, and the result of the numerical operator is a value of type double. (If the other operand is not a double, it is first widened to type double by numeric promotion.)

A.3.2 Reference Types

There are three kinds of reference types: class types, interface types, and array types. Reference values are pointers to objects (instances of classes or arrays), or a null reference, which specifies no object. Objects are created dynamically using class instance creation expressions and array creation expressions.

Operators on reference values are:

- Field access
- Method invocation
- Cast
- String concatenation
- instanceof
- Reference equality operators

The standard class Object is the base superclass of all classes. A variable of type Object can hold a reference to any object, including an instance of a class or an array. See Section A.12 for the specification of class Object.

A.3.3 Variables

A variable is a named storage location that has an associated type (either primitive [A.3.1] or reference [A.3.2]). A variable's value is changed by an assignment operator.

iJava is *strongly typed*: it guarantees that a variable always holds a value compatible with its type. Primitive-type variables always hold a value whose type exactly matches the type of the variable. Reference-type variables hold either a null reference or a reference to an object whose class is assignment compatible with the variable's type.

There are seven kinds of variables:

1. A *class variable* is a variable defined with the keyword static inside a class declaration.
2. An *instance variable* is a variable defined without the keyword static inside a class declaration.
3. An *array component* is an unnamed variable that is a member of an array.

4. A *method parameter* is a formal parameter in a method definition.

5. A *constructor parameter* is a formal parameter in a constructor definition.

6. An *exception-handler parameter* is defined by a catch clause of a try statement.

7. A *local variable* is defined by local variable declaration statements.

iJava guarantees that every variable has a value before its value is accessed by a program. Variables are initialized as follows:

- Class variables, instance variables, and array components are initialized with a default value as follows:
  - Numeric types are initialized to 0
  - Boolean type is initialized to false
  - Reference types are initialized to null
- Method and constructor parameters are initialized to the corresponding value of the invoker of the method or constructor
- Exception handler parameters are initialized to the exception object that was thrown
- Local variables must be explicitly initialized through initialization or assignment

A.4 Conversions and Promotions

Expressions in iJava programs produce values with a given type. The type produced by an expression must be compatible with the context in which the expression occurs. iJava provides both implicit and explicit mechanisms for converting the types of expressions to allow them to be used in various contexts. This section describes the types of conversions and the contexts in which they can be used.

A.4.1 Kinds of Conversions

This section describes the kinds of conversions supported in iJava.

Widening primitive conversions: Any integer type may be converted to a larger integer type, or to double. Except in the case of long to double, these conversions never cause loss of precision.

Narrowing primitive conversions: The conversion of an integer type to a smaller integer type, or double to an integer type, is a narrowing conversion. These conversions may cause loss of magnitude or precision. In the case of narrowing conversions from an integer to an integer type, the result is computed by discarding all but the lowest n bits of the source integer, where n is the number of bits of the target type. In the case of narrowing conversions from double to an integer type, the result is computed as follows:

1. First, the double value is converted to a long, if the target is a long, otherwise to an int. This conversion has one of the following results:
   - If the value is NaN, the result is 0.
   - If the value is too large or too small to be represented in the target type, the result is the largest or smallest possible value in the target type, respectively.
   - Otherwise, the value is converted to the target type, rounded towards 0.
2. Second, if the target type is byte, char, or short, a narrowing conversion is applied to the int resulting from the first step.

Widening reference conversions: These conversions apply to reference types. Any reference type can be converted to Object. The following widening reference conversions are also allowed:

- From a class type S to a class type T, when S is a subclass of T
- From a class type S to an interface type K, when S implements K
- From the null type to any reference type
- From any interface type J to any interface type K, if J is a subinterface of K

Narrowing reference conversions: These conversions apply to reference types:

- From a class type S to a class type T, when S is a superclass of T
- From a class type S to an interface type K, when S does not implement K
- From type Object to any array or interface type

These conversions require a runtime test to determine whether the reference value being converted is an instance of the target type (or a subclass of it). A ClassCastException is thrown if it is not.

String conversions: There is a conversion to type String from every type.

A.4.2 Assignment Conversions

An assignment statement performs assignment conversion to convert the type of the expression to the type of the variable. An expression is assignment compatible with the type of the variable if it can be converted to the variable's type with a widening primitive conversion or a widening reference conversion (see Section A.4.1). Also, if the expression is a constant of type int, and a narrowing primitive conversion to the type of the variable would cause no loss of data, such a conversion is permitted.

A.4.3 Method Invocation Conversions

A method invocation performs method invocation conversions to convert the the types of the parameter expressions to the types of the formal parameters. A parameter expression is compatible with the type of the corresponding formal parameter if it can be converted to the variable's type with a widening primitive conversion or a widening reference conversion (see Section A.4.1). Note that, unlike assignment conversions, no narrowing primitive conversions are supported for method invocation, to simplify the method overloading rules.

A.4.4 Casting Conversions

Casting conversion is applied to the operand of the cast operator (see Section A.11.12). A cast can do any permitted conversion other than a string conversion.

A.4.5 Numeric Promotions

Numeric promotions occur in the context of arithmetic operators. They involve widening primitive conversions. Unary numeric promotions occur in conjunction with unary arithmetic operators; binary numeric promotions occur in conjunction with binary arithmetic operators.

Unary numeric promotion: If the operand is of type byte, short, or char, a widening conversion is applied to convert it to type int.

Binary numeric promotion: This conversion is applied to a pair of operands, to convert them to the same type before the operation takes place. It occurs as follows. If either operand is of type double, the other is converted to double. Otherwise, if either operand is a long, the other is converted to a long; otherwise, both operands are converted to int.

A.5 Packages and Compilation Units

iJava programs are organized into units called *packages*. A package contains a collection of compilation units that comprise class and interface declarations. Packages have a hierarchical organization: a subpackage Q of a package P has the fully qualified name $P.Q$.

A.5.1 Compilation Units

A compilation unit has three parts:

- A package declaration, with the fully qualified name of the package to which the compilation unit belongs
- import declarations allow references to classes and interfaces from other packages by their simple names
- A class or interface declaration

Each of these parts is optional.

All iJava classes and interfaces must be declared with the public keyword, indicating that they are accessible to all packages, not just the one in which they are declared.

A.5.2 Import Declarations

An import declaration allows code in a compilation unit to reference classes and interfaces from other packages by their simple names. There are two forms of an import declaration:

- A *single-type import declaration* specifies the fully qualified name of a class or interface, making it available via its simple name to the code within the compilation unit. It has the following syntax:

  `import` *TypeName* `;`

  The *TypeName* is a fully qualified name of a class or interface type.

- A *wildcard import declaration* specifies the name of a package from which all classes or interfaces are imported into the current namespace. It has the following syntax:

  `import` *PackageName* `. * ;`

  The *PackageName* is the name of the package whose classes and interfaces are to be imported.

Each compilation unit automatically imports each of the classes and interfaces declared in the predefined package java.lang.

A.6 Classes

A class declaration binds an identifier to a class type, establishing a new reference type with its implementation. In this section, we discuss the syntax and semantics of iJava class declarations.

A.6.1 Class Declarations

A class declaration has the following form:

```
ClassDeclaration:
    public [abstract] class Identifier [TypeParameters] [Super]
        [Interfaces] ClassBody

TypeParameters:
    < TypeName {, TypeName} >

Super:
    extends ClassType

ClassType:
    TypeDeclSpecifier [TypeArguments]

Interfaces:
    implements InterfaceType {, InterfaceType}

InterfaceType:
    TypeDeclSpecifier [TypeArguments]
```

TypeArguments:

< Type { , Type } >

iJava classes must be defined as public, indicating their global scope. iJava classes which contain abstract methods must be defined as abstract. iJava classes may declare that they implement interfaces via the implements clause, and may declare that they subclass a parent class via the extends clause. Classes which do not name a parent class inherit from the Object class.

A generic class declaration specifies *TypeParameters*, which define names that may be used as placeholders for data types through the body of the class declaration. A class declaration that implements generic interfaces or extends a generic parent class may provide concrete *TypeArguments* to supply values for the type parameters of the class or interface being extended or implemented.

A class body may contain fields, method declarations, and constructors (see Section A.6.3).

A.6.2 Abstract Classes

An abstract class is one that contains an abstract method. There are three ways a class may contain an abstract method:

- It may declare an abstract method.
- It may inherit an abstract method from an ancestor class.
- It may fail to implement a method specified by a superinterface, either directly or through inheritance from a superclass

Such a class must be marked with the abstract modifier. Abstract classes may not be instantiated: they serve only as superclasses for other abstract classes or for concrete classes.

A.6.3 Class Members

The body of a class has the following form:

ClassBody:

{ { ClassBodyDeclaration } }

ClassBodyDeclaration:

FieldDeclaration

MethodDeclaration

ConstructorDeclaration

The members of a class include all of the following:

- Methods and fields declared within the class
- Members (other than constructors) inherited from the superclass
- Members inherited from superinterfaces

These member declarations must be marked with public, private, or protected modifiers (iJava does not have Java's default package access type). The modifiers have the effect of restricting access to the member from code outside the class as follows:

- public members are accessible without restriction.
- private members are accessible only within the class in which they are defined.
- protected members are accessible within the class and its subclasses.

A.6.4 Field Declarations

Field declarations have the following form:

```
FieldDeclaration:
  VisibilityModifier {FieldModifier} Type VariableDeclarators ;

FieldModifier:
  static
  final

VariableDeclarators:
  VariableDeclarator {, VariableDeclarator}

VariableDeclarator:
  Identifier [= VariableInitializer]
```

VariableInitializer:

Expression

ArrayInitializer

A field declaration consists of a visibility modifier, other field modifiers, a data type and an identifier. An initializer expression may provide an initial value for the field. The scope of a field declaration is the entire body of the class. However, field initializers may not reference fields that appear after the initializer in the class.

Methods and fields within a class may have the same name, but two fields within a class may not have the same name. Further, a field declared within a class may not have the same name as a public or protected field inherited from a superclass. This is a departure from Java, which allows shadowing of fields.

Class fields may be instance members, which are instantiated separately for each instance of the class, or class members (denoted with the static modifier), which are instantiated once and shared by all instances of the class. Class members may be marked final, specifying that their value may not be changed after initialization; in iJava, instance members may not be marked final.

The initialization of class fields has the semantics of assignment statements. Class members are initialized once, before the class is instantiated and before any invocation to a static member method. Instance members are initialized upon class instantiation; the variable initializer is evaluated each time the class is instantiated.

A.6.5 Method Declarations

Method declarations have the following form:

MethodDeclaration:

MethodHeader MethodBody

MethodHeader:

{MethodModifiers} ResultType MethodDeclarator

MethodModifiers:

public

```
    protected
    private
    static
    abstract

ResultType:
    Type
    void

MethodDeclarator:
    Identifer ( [FormalParameterList] )

FormalParameterList:
    FormalParameter {, FormalParameter}

FormalParameter:
    Type VariableDeclaratorId
```

A method declaration consists of a return type, an identifier, a parenthesized list of parameters, and, optionally, a method body. If the body is not present, the method must be marked with the abstract modifier.

The scope of a method's formal parameters is the entire body of the method. A method's formal parameter names may not be reused as local variables.

Method names may be overloaded: two methods in a class may have the same name, as long as they can be distinguished by their signatures.

A method declared with the static modifier must not contain the keywords this or super in its body. It is invoked without a reference to a particular object. A method declared without the static modifier is invoked on a reference to an object, which becomes the context for resolving references to methods and fields via the *this* and super keywords.

A method with a void return type must not contain a return statement that has an expression. However, a method with a non-void return type must contain a return statement with an expression. The value of the expression becomes the value of the method invocation expression in the caller's context.

A.7 Interfaces

An interface declaration binds a name to a new reference type. The type specifies method signatures that govern the methods that may be invoked on values of its type, but does not specify implementations for the methods.

Interfaces in iJava provide a subset of the capabilities found in Java interfaces. The chief difference is that iJava interfaces, unlike Java interfaces, may not have constants or specify default method implementations. An interface declaration has the following form:

```
InterfaceDeclaration:
    public interface Identifier [TypeParameters]
        [ExtendsInterfaces] InterfaceBody


TypeParameters:
    < TypeName {, TypeName} >
```

An iJava interface must be declared public, and is globally accessible.

An interface may specify one or more type parameters in angle brackets. This indicates that the interface is generic.

An interface may specify that it extends one or more interfaces:

```
ExtendsInterfaces:
    extends TypeName {, TypeName}
```

This has the effect of including in the interface all methods declared by its superinterfaces.

An interface may contain one or more method signatures, but may not contain method bodies:

```
InterfaceBody:
    { {AbstractMethodDeclaration} }


AbstractMethodDeclaration:
    ResultType MethodDeclarator ;
```

Method declarations in interfaces are implicitly public and abstract.

An interface may contain more than one method with the same name, as long as the signatures of the methods distinguish them.

A.8 Arrays

iJava arrays are dynamically created objects that contain a collection of unnamed components that are referenced using a non-negative index. The collection may be empty. The length of the array corresponds to the number of components.

All components in an array have the same *component type*. An array with component type T has a type denoted $T[]$. The component type itself may be an array type; the type of the leaf components of such an array is known as the *element type*.

A variable of an array type holds a reference to an array. The initial value of such a variable, if unspecified, is the null reference. Alternatively, an initializer may instantiate an array and assign a reference to it to the variable.

A.8.1 Array Access

The components of an array of length n may be accessed with an array access expression, as discussed in Section A.11.10. The type of the expression must be one of the following integer types: byte, short, int, char.

A.8.2 Array Creation

Arrays are created using an array initializer, or an array creation expression. Array creation expressions are discussed in Section A.11.7. Array initializers are a sequence of expressions in curly brackets, as follows:

```
ArrayInitializer:
    { {VariableInitializer ,} }

VariableInitializer:
    Expression
    ArrayInitializer
```

The expressions in the array initializer are evaluated from left to right, and are assigned in sequence to the components of the array.

A.8.3 Array Members

The members of an array include:

- A read-only property, int length, that specifies the number of components in the array
- All members inherited from class Object

A.9 Exceptions

Exceptions are used in iJava to report errors and to facilitate the transfer of control to a nonlocal error handler. The main difference from Java is that iJava supports only unchecked exceptions.

Exceptions in iJava are thrown for one of three reasons:

- An application error occurred, such as an attempt to index an array with an integer that does not denote a valid array component.
- The language implementation detected an error such as out of memory or stack overflow.
- A throw statement in an iJava program raised an exception.

iJava exceptions are instances of the class Error or RuntimeException.

A.9.1 Exception Handling

When an exception is thrown, control transfers to the nearest compatible catch clause of a try statement that encloses the code that triggered the exception, or that encloses a method invocation in the call stack. A catch clause is compatible with an exception if the exception is an instanceof the declared exception parameter type. If no catch clause is in the scope of the thrown exception, the program terminates.

A.9.2 Exception Classes

The exception class hierarchy in iJava is rooted at a class Throwable. Subclasses of Throwable include Exception and Error. Exception has a subclass, RuntimeException. Note that iJava permits only instances of RuntimeException and Error and their subclasses to be thrown, as iJava does not support checked exceptions.

A.10 Blocks and Statements

A.10.1 Local Variable Declaration Statements

A local variable declaration statement defines one or more variables.

```
LocalVariableDeclarationStatement:
    Type VariableDeclarator {, VariableDeclarator}

VariableDeclarator:
    Identifier [= VariableInitializer]

VariableInitializer:
    Expression
    ArrayInitializer
```

The variable is defined with the specified type and identifier, and the scope of the definition is the remainder of the block within which the variable declaration statement appears.

If a VariableInitializer is provided, the value of the expression or array initializer, whose type must be checked at compile time to be assignment compatible with the variable's type, is used to initialize the value of the variable. Initialization expressions are evaluated in the order in which they appear. If no expression or array initializer is provided, the compiler must verify that the variable is explicitly assigned a value before its value is used in an expression.

Local variables are not permitted to shadow method parameters or other local variables.

A.10.2 Empty Statement

The empty statement does nothing.

```
EmptyStatement:
    ;
```

A.10.3 Assignment Statement

The assignment statement stores a value in a variable:

```
AssignmentStatement:
    LeftHandSide AssignmentOperator Expression ;
```

```
LeftHandSide:
    ExpressionName
    FieldAccess
    ArrayAccess

AssignmentOperator: one of
    = *= /= %= += -= <<= >>= >>>= &= ^= |=
```

LeftHandSide must denote a variable (see Section A.3.3). The type of *Expression* is checked at compile time and must be assignment compatible with the variable denoted by *LeftHandSide*.

Standard assignment: Runtime execution of the standard assignment statement proceeds as follows. First, *LeftHandSide* is evaluated to determine which variable, field, or array component is the target of the assignment. If this evaluation completes successfully with no runtime exception, the *Expression* is evaluated to produce the value to be stored in the variable. Finally, the resulting value is converted to the type of *LeftHandSide* and stored into the variable.

Augmented assignment: The augmented assignment operators other than += require that both operands have a primitive type. The += operator allows any type of expression on the right-hand side, if the left-hand operand has type String. An augmented assignment expression of the form `Expr1` *op*`= Expr2` is equivalent to `Expr1 = (T)((Expr1)` *op* `(Expr2))`, except that `Expr1` is evaluated only once.

Runtime evaluation of the augmented assignment statement proceeds as follows. First, *LeftHandSide* is evaluated to determine which variable, field, or array component is the target of the assignment, and to produce the value for the left-hand side of the binary operator. If this evaluation completes successfully with no runtime exception, the *Expression* is evaluated to produce the value for the right-hand side of the binary operator. The binary operation is performed, and the result of the binary operation is converted to the type of *LeftHandSide*. Finally, the resulting value is stored into the variable.

A.10.4 Expression Statement

Certain expressions can be used as statements:

```
ExpressionStatement:
```

```
    StatementExpression ;

StatementExpression:
    Assignment
    MethodInvocation
    ClassInstanceCreationExpression
```

Expression statements are executed by evaluating the expression and discarding the result.

A.10.5 If Statement

An if statement conditionally executes a choice of up to two statements:

```
IfStatement:
    if ( Expression ) Statement [else Statement]
```

The statement is executed by first evaluating the Expression, which must be boolean-valued. If the value is true, the first *Statement* is executed. If the value is false, and the else part is present, the second *Statement* is executed.

A.10.6 While Statement

The while statement provides iJava's loop construction:

```
WhileStatement:
    while ( Expression ) Statement
```

The statement is executed as follows:

1. The *Expression*, which must be boolean-valued is evaluated. If the value is false, the *Statement* is not executed, and the loop terminates.
2. If the value is true, the *Statement* is executed.
3. Go to step 1.

The *Statement* may be prematurely terminated with a break or continue statement. If a break statement is encountered, the loop terminates immediately. If a continue statement is encountered, the loop resumes execution beginning with the evaluation of the *Expression*.

A.10.7 Return Statement

The return statement is used to terminate a method:

```
ReturnStatement:
    return [Expression] ;
```

A return statement transfers control from the current method back to the caller at the point of invocation of the current method. If the return statement is located in a void method or constructor, it must not contain an expression; if it is located in a non-void method, it must contain an expression. In the latter case, the *Expression* is evaluated, and its value becomes the value of the method invocation expression that initiated the current method.

A.10.8 Throw Statement

The throw statement is used to raise an exception:

```
ThrowStatement:
    throw Expression ;
```

The type of the Expression must be class Error or RuntimeException or a subclass of those classes. The statement execution begins with evaluation of the Expression. Then, control is transferred to the nearest compatible catch block associated with a dynamically-enclosing try statement.

A.10.9 Try Statement

The try statement is used to establish a context for exception handling:

```
TryStatement:
    try Block CatchClause { CatchClause }


CatchClause:
    catch ( FormalParameter ) Block


FormalParameter:
    Type Identifier
```

The statements in *Block* are executed sequentially, until all the statements complete successfully, or an exception is thrown. If an exception is raised during execution of the *Block*, control

transfers to the first *CatchClause* with a *FormalParameter* that is assignment compatible with the thrown exception object. If no catch clause has an assignment compatible parameter, the search for a handler continues with the next dynamically enclosing try statement.

A.11 Expressions

This section describes the various expressions available in iJava.

A.11.1 Type of Expressions

Every expression, except for invocations of void methods, has a type, known at compile time, and denotes a value, computed at runtime. The rules for determining the compile-time type of expressions are covered below. The runtime value is guaranteed to be assignment compatible with the type of the expression.

A.11.2 Evaluation Order

The order of evaluation of operands in an expression is undefined in iJava. Further, the order of evaluation of parameters in a method invocation expression is undefined.

A.11.3 Primary Expressions

The following are the simplest expressions from which others are constructed: array creation, literals, field and method invocations, array access, and parenthesized expressions.

```
Primary:
    ArrayCreationExpression
    Literal
    this
    ( Expression )
    ClassInstanceCreationExpression
    FieldAccess
    MethodInvocation
    ArrayAccess
    Identifier
```

An *Identifier* that occurs within the scope of a parameter or local variable with its name denotes the type and value of the corresponding parameter or local variable.

The meaning of other primary expressions is given in sections that follow.

A.11.4 Literal Expressions

Literal expressions have types as defined here:

- Integer literals have type int.
- Floating point literals have type double.
- Boolean literals have type boolean.
- String literals have type String.
- The null literal has the null type, and its value is the null reference.

A.11.5 The this Expression

The this keyword may be used in the body of an instance method or constructor. It denotes the reference of the object on which the method is invoked, or of the object being constructed. Its compile time type is the type of the class in which it occurs; at runtime, its type may also be a subtype of its compile time type.

A.11.6 Class Instance Creation Expressions

A class instance creation expression creates instances of classes:

```
ClassInstanceCreationExpression:
    new Identifier [TypeArguments] ( [ArgumentList] )
```

The *Identifier* must be the name of a class. If the class is generic, *TypeArguments* must be supplied to provide types for the generic class type parameters. The *ArgumentList* provides values to initialize the parameters of a constructor in the class being instantiated.

The expression is evaluated at runtime by allocating space for a new class instance, and initializing it using a constructor. The constructor used to initialize the instance is selected using the supplied *ArgumentList*, using the same rules as for method invocations (see Section A.11.9).

A.11.7 Array Creation Expressions

An array creation expression is used to create instances of arrays:

```
ArrayCreationExpression:
    new PrimitiveType DimExprs {[ ]}
    new TypeName DimExprs {[ ]}

DimExprs:
    DimExpr {DimExpr}

DimExpr:
    [ Expression ]
```

The resulting array holds elements whose type are specified by the supplied *PrimitiveType* or *TypeName*. *TypeName* may be either a class or interface type. The type of each *Expression* in *DimExpr* must be an integer type other than long: byte, short, int, or char.

At runtime, the array creation expression is evaluated first by checking the values of the *DimExpr* expressions; if any is less than 0, a NegativeArraySizeException is thrown. Next, space is allocated for the array. If insufficient space is available, an OutOfMemoryError exception is thrown. Next, if there is a single *DimExpr* expression, each component of the array is initialized to its default value. If there are multiple *DimExpr* expressions, each level of the array corresponding to a *DimExpr* expression is initialized to create the arrays of arrays.

A.11.8 Field Access Expressions

A field access expression denotes a field of an object or array:

```
FieldAccess:
    Primary . Identifier
```

The *Identifier* denotes a field within the object or array denoted by *Primary* or *ClassName*. The type of the expression is the declared type of the field, and its runtime value is the value of the field.

If the field is not static and the value of *Primary* is null, the resulting behavior is undefined. This is a departure from Java, where a NullPointerException is thrown in this case.

A.11.9 Method Invocation Expressions

A method invocation expression causes the code associated with a particular method to be executed:

```
MethodInvocation:
    Identifier ( [ArgumentList] )
    Primary . Identifier ( [ArgumentList] )
    super . Identifier ( [ArgumentList] )

ArgumentList:
    Expression {, Expression}
```

The process of determining which method is invoked involves several steps, some performed at compile time, others at runtime. The first step, performed at compile time, is to determine which class to search.

- If the form is *Identifier*, the class to search is the class containing the method invocation.
- If the form is *Primary . Identifier*, the class to search is the type of *Primary*.
- If the form is *super . Identifier*, the class to search is the superclass of the class containing the method invocation.

The second step involves determining which method in the target class that is accessible has a declaration signature that is the closest match to the argument list. A method is *accessible* if its access modifier (public, private, or protected) provides visibility to the method from the point of the method call. A declaration signature is a match to the argument list if each expression in the argument list is assignment compatible with the corresponding formal parameter in the signature. For the details of this step of the search, see section 15.12.2 of *The Java Language Specification* [20]. If no suitable method is found, a compile time error results.

The third step involves checking whether the chosen method in the target class is appropriate. There are two cases to consider:

- If the form is *Identifier*, and the method is invoked from a static method, the chosen method must be static.

- If the form is *Primary* . *Identifier*, and *Primary* denotes a class name, the chosen method must be static.

The remaining steps occur at runtime. First, if the target method is not static, a target reference is computed. In the case of the first and third alternatives in the grammar above, the target reference is this; otherwise, it is the value of *Primary*.

Next, the method arguments are evaluated. The evaluation order of the arguments is undefined in iJava.

If the value of the target reference is null, the ensuing behavior is undefined. This is a departure from Java, where a NullPointerException is thrown in the case of a null target reference at this point.

Next, if the target method is static, the target method is invoked. If the target method is not static, a runtime dynamic lookup is performed to locate the target method, beginning with a class *S*, which is the runtime type of the target reference, or in the case of the *super* case, the superclass of the target reference.

- If *S* contains a method that matches the target method signature, this method is invoked.
- Otherwise, the search continues with the superclass of *S*.

The search is guaranteed to terminate because of the checks performed at compile time, specified above.

A.11.10 Array Access Expressions

An array access expression references a component of an array:

```
ArrayAccess:
    Primary [ Expression ]
```

Primary may be any primary expression, other than an array creation expression, whose type is an array type. Suppose the type is *T[]*. Then the type of the array access expression is *T*, and the resulting value is the component of the array denoted by *Primary*, at position denoted by *Expression*. The type of *Expression* must be an integer type (either byte, short, char, or int, but not long).

At runtime, the *Primary* expression is evaluated to produce a reference to an array. If the value of *Primary* is null, the behavior is undefined.

Next, the *Expression* is evaluated to produce the index. If the index is not in the range of 0 .. *length* - 1, where *length* is the size of the array, an ArrayIndexOutOfBounds exception is thrown.

Finally, the value of the component is computed.

A.11.11 Unary Operators

The unary operators include +, -, !, ∼, and the cast operator. Expressions with unary operators group right to left.

Unary Plus (+): The type of the operand must be numeric, or a compile error results. The value of the expression is the promoted value of the operand; the type of the expression is its promoted type.

Unary Minus (-): The type of the operand must be numeric, or a compile error results. The value of the expression is the negation of the promoted value of the operand; the type of the expression is its promoted type. In the case of an overflow (ex. the negation of the smallest possible negative integer), the result is undefined.

Bitwise Complement (∼): The type of the operand must be integral, or a compile error results. The value of the expression is the bitwise complement of the promoted value of the operand: the type of the expression is its promoted type.

Logical Complement (!): The type of the operand must be boolean, or a compile error results. The value of the expression is the logical negation of the operand; the type of the expression is boolean.

A.11.12 Cast Expressions

A cast expression is used to convert a numeric operand to a different numeric type, or to confirm the type of a reference operand:

```
CastExpression:
    ( PrimitiveType [Dims] ) Expression
    ( ReferenceType ) Expression
```

The type of a cast expression is *PrimitiveType* or *ReferenceType*. The value of the cast expression is the result of a casting conversion (see Section A.4.4).

A primitive value may not be cast to a *ReferenceType*, and vice versa. These errors are detected at compile time. A reference type cast conversion that cannot be proven correct by the compiler is verified at runtime; a ClassCastException is thrown in the event such a conversion fails at runtime.

A.11.13 Arithmetic Operators

The operators +, -, *, /, and % (integer modulus) are the binary arithmetic operators. They are left associative, and have the usual mathematical precedence.

The type of an arithmetic expression is the promoted type of its operands; the value is the result of performing the operation on the promoted values of the operands. If overflow results, the result is not defined.

Integer division truncates the result (rounds towards zero). Integer division by zero causes an ArithmeticException to be thrown; floating point division by zero results in the value NaN.

The modulus operator % accepts only integral operands (a departure from Java, in which floating point operands are also supported for this operator). It produces a result for operands a and b such that `(a/b)*b+(a%b)` is equal to a. If the divisor is 0, an *ArithmeticException* is thrown.

If the type of either operand of the + operator is String, then string concatenation is performed, and the type of the result is String.

String concatenation: String concatenation results in a string in which the characters of the first operand are immediately followed by the characters of the second operand. When only one operation of the + operator is String, the other operand is converted to a String at runtime. The conversion of a value to String is as follows:

- The conversion of a primitive value to a String must have the same effect as the invocation of the toString() method on a newly constructed instance of a wrapper class that corresponds to the primitive value.
- The conversion of a reference value to a String is performed by invoking toString() on the reference object; in the case in which the reference value is null, the resulting string is the value "null".

A.11.14 Shift Operators

The shift operators include left shift <<, signed right shift >>, and unsigned right shift >>>. They are syntactically left-associative. The left-hand operand specifies the value to be shifted; the right-hand operand, the shift distance.

Each operand must be an integer type; unary numeric promotion is performed separately on each operand, and the type of the expression is the promoted type of the left-hand operand.

If the promoted type of the left-hand operand is int, only the rightmost 5 bits of the right-hand operand are used to determine the shift distance, yielding a possible distance of 0-31. If the promoted typehe left-hand operand is long, only the rightmost 6 bits of the right-hand operand are used to determine the shift distance, yielding a possible distance of 0-63.

The value resulting from the shift operators is as follows:

- The value of `n << s` is n left-shifted s bit positions.
- The value of `n >> s` is n right-shifted s bit positions, with sign extension.
- The value of `n >>> s` is n right-shifted s bit positions, with zero extension.

A.11.15 Relational Operators

The relational operators are <, <=, >, >=. Syntactically, they are non-associative. The resulting type of a relational operator is boolean.

The relational operators require operands of numeric type. After binary numeric promotion is performed on the operands, the appropriate comparison is computed as follows:

- The < operator produces true if the left-hand operand is numerically less than the right-hand operand.
- The <= operator produces true if the left-hand operand is numerically less than or equal to the right-hand operand.
- The > operator produces true if the left-hand operand is numerically greater than the right-hand operand.
- The >= operator produces true if the left-hand operand is numerically greater than or equal to the right-hand operand.

A.11.16 Equality Operators

The operators == and != test for equality of their operands. Syntactically, they are left-associative. They may be used to compare two operands of numeric type; two operands of boolean type; or two operands of reference type. Other combinations are not legal. The type produced by equality operators is boolean in all cases.

The value produced by `a!=b` is always the same as the value produced by `!(a==b)`. Thus, in the following discussion, only the value produced by the == operator is specified. There are three cases to consider for ==:

- If the operands are numeric, binary numeric promotion is performed, and the resulting values are compared for equality. The value true is produced if the two values are numerically equal.
- If the operands are boolean, the value true is produced if the two values are the same boolean value.
- If the operands are a reference type, the comparison is legal at compile time only if it is possible to use a casting conversion to convert the type of one operand to the other. At runtime, the value true is produced if both operands are null, or both refer to the same object instance.

A.11.17 instanceof Operator

The instanceof operator tests if a reference value can be cast to a specified reference type. The result type of the operator is boolean.

The left-hand operand of instanceof must be a reference type; the right-hand operand must be the name of a reference type. The result of the instanceof operator is true if the left-hand operand is non-null, and the reference could be cast to the type specified by the right-hand operand. A compile error results if an attempt to cast the left-hand operand to the type specified by the right-hand operand would result in a compile error.

A.11.18 Bitwise and Logical Operators

The bitwise and logical operators include and (`&`), or (|), and exclusive or (`^`). They accept two operands of either integer type or boolean type. Other combinations result in a compile error.

Bitwise operators: When both operands are of an integer type, binary numeric promotion is performed on the operands, and the bitwise operation corresponding to the operator is performed on the promoted values. The result is the promoted integer type.

Logical operators: When both operands are of boolean type, the result has type boolean. If the operator is &, the result is true if both operands are true. If the operator is |, the result is true if either operand is true. If the operator is ^, the result is true if the operand values are different.

A.11.19 Conditional Operators

The conditional operators include conditional and (&&), conditional or (||), and the ternary operator (?:).

Conditional and: The types of the operands of conditional and must both be boolean. At runtime, the left-hand operand is evaluated. If it is false, the result of the operation is false, and the right-hand operand is not evaluated. Otherwise, the right-hand operand is evaluated, and the result of the operation is true if both operands were true, and false otherwise.

Conditional or: The types of the operands of conditional or must both be boolean. At runtime, the left-hand operand is evaluated. If it is true, the result of the operation is true, and the right-hand operand is not evaluated. Otherwise, the right-hand operand is evaluated, and the result of the operation is false if both operands were false, and true otherwise.

Ternary: The ternary operator is syntactically right-associative, and has three operands:

```
ConditionalExpression:
    Expression ? Expression : ConditionalExpression
```

The type of the first expression must be boolean. The second and third operands must be either both numeric, or both boolean, or both a reference type. At runtime, the first expression is evaluated; if it is true, the second expression is evaluated, and its value becomes the value of the entire expression; the third expression is not evaluated. If the first expression is false, the third expression is evaluated, and its value becomes the value of the entire expression; the second expression is not evaluated in this case.

A.12 Class Library

This section defines the classes in the java.lang package, which comprise the iJava standard library.

A.12.1 java.lang.Object

The class Object is the single root of the class hierarchy. All objects, including arrays, implement the methods of this class.

```
public class Object {
        public String toString();
        public boolean equals(Object obj);
        public int hashCode();
}
```

- **String toString()**: Returns a String representation of the state of the receiver. Designed to be overridden, the default implementation returns an unspecified value.
- **boolean equals(Object obj)**: The default implementation returns true if this == obj is true. The contract is intended to implement an equivalence relation between objects that is reflexive, symmetric, transitive, and consistent.
- **int hashCode()**: This method is intended to support hash tables. The default implementation returns 0.

A.12.2 java.lang.Boolean

The class Boolean is a wrapper class; instances of this class wrap individual boolean values, to permit them to be used in contexts where a reference type is required.

```
public class Boolean {
        public Boolean(boolean value);
        public boolean booleanValue();
        public static final Boolean TRUE, FALSE;
}
```

- **Boolean(boolean value)**: Initializes an instance so that it represents the indicated *value*.
- **boolean booleanValue()**: Returns the value with which this instance was initialized.
- **static Boolean TRUE**: An instance of Boolean that wraps the value true.
- **static Boolean FALSE**: An instance of Boolean that wraps the value false.

A.12.3 java.lang.Character

The class Character is a wrapper class; instances of this class wrap individual char values, to permit them to be used in contexts where a reference type is required.

```
public class Character {
        public Character(char value);
        public char charValue();
}
```

- **Character(char value)**: Initializes an instance so that it represents the indicated *value*.
- **char charValue()**: Returns the value with which this instance was initialized.

A.12.4 java.lang.Number

The class Number has subclasses Integer, Long, and Double. It provides methods to extract representations of the primitive value wrapped by the subclasses.

```
public abstract class Number {
        public abstract int intValue();
        public abstract long longValue();
        public abstract double doubleValue();
}
```

- **double doubleValue()**: Returns the value with which this instance was initialized after converting it to double.
- **int intValue()**: Returns the value with which this instance was initialized after converting it to int.

- **long longValue()**: Returns the value with which this instance was initialized after converting it to long.

A.12.5 java.lang.Double

The class Double is a wrapper class; instances of this class wrap individual double values, to permit them to be used in contexts where a reference type is required.

```
public class Double extends Number {
        public Double(double value);
        public int intValue();
        public long longValue();
        public double doubleValue();
}
```

- **Double(double value)**: Initializes an instance so that it represents the indicated *value*.
- **double doubleValue()**: Returns the value with which this instance was initialized.
- **int intValue()**: Returns the value with which this instance was initialized after converting it to int.
- **long longValue()**: Returns the value with which this instance was initialized after converting it to long.

A.12.6 java.lang.Integer

The class Integer is a wrapper class; instances of this class wrap individual int values, to permit them to be used in contexts where a reference type is required.

```
public class Integer extends Number {
        public Integer(int value);
        public int intValue();
        public long longValue();
        public double doubleValue();
        public static int parseInt(String s);
```

```
        public static int parseInt(String s, int radix);
}
```

- **Integer(int value)**: Initializes an instance so that it represents the indicated *value*.
- **double doubleValue()**: Returns the value with which this instance was initialized.
- **int intValue()**: Returns the value with which this instance was initialized after converting it to int.
- **long longValue()**: Returns the value with which this instance was initialized.
- **static int parseInt(String s)**: Returns the value that results from interpreting s as a signed base 10 integer.
- **static int parseInt(String s)**: Returns the value that results from interpreting s as a signed integer in the base specified by *radix*.

A.12.7 java.lang.Long

The class Long is a wrapper class; instances of this class wrap individual long values, to permit them to be used in contexts where a reference type is required.

```
public class Long {
        public Long(long value);
        public int intValue();
        public long longValue();
        public double doubleValue();
}
```

- **Long(long value)**: Initializes an instance so that it represents the indicated *value*.
- **double doubleValue()**: Returns the value with which this instance was initialized after converting it to double.
- **int intValue()**: Returns the value with which this instance was initialized after converting it to int.
- **long longValue()**: Returns the value with which this instance was initialized.

A.12.8 java.lang.String

The class String provides iJava's String data type and operations.

```
public class String {
        public String();
        public String(char[] chars);
        public boolean equals(Object anObject);
        public int hashCode();
        public int length();
        public char charAt(int index);
        public char[] toCharArray();
        public boolean equalsIgnoreCase(String anotherString);
        public int compareTo(String anotherString);
        public int indexOf(int ch);
        public int indexOf(int ch, int fromIndex);
        public int indexOf(String str)
        public int indexOf(String str, int fromIndex)
        public String substring(int beginIndex);
        public String substring(int beginIndex, int endIndex);
        public String concat(String str);
        public String toLowerCase();
        public String toUpperCase();
        public static String valueOf(Object obj);
        public static String valueOf(boolean b);
        public static String valueOf(char c);
        public static String valueOf(int i);
        public static String valueOf(long l);
        public static String valueOf(double d);
}
```

- **String()**: Initializes an instance so that it represents an empty string.

- **String(char[] chars)**: Initializes an instance so that it represents the sequence of characters in chars.
- **boolean equals(Object anObject)**: Overrides Object's equals() to return true when this instance contains the same sequence of characters as anObject, if anObject is a String.
- **int hashCode()**: Overrides Object's hashCode() to return a hash value computed using this string's characters.
- **int length()**: Returns the number of characters in this string.
- **char charAt(int index)**: Returns the character at position index in this String. Throws IndexOutOfBoundsException if index is not in the range 0 .. length() - 1.
- **char[] toCharArray()**: Returns an array containing the characters of this string.
- **boolean equalsIgnoreCase(String anotherString)**: Returns true if the sequence of characters in this string is equal to that of anotherString, after both sequences have been converted to the same capitalization.
- **int compareTo(String anotherString)**: Returns a negative integer if the sequence of characters in this string lexicographically precedes the sequence of anotherString; a positive integer if this string lexicographically follows anotherString; 0 if the two strings are identical.
- **int indexOf(int ch)** and its overloads: Returns the position of a character or string in the sequence of characters in this string. The variants that take fromIndex permit the starting position of the search to be specified.
- **String substring(int beginIndex, int endIndex)** and overload: Returns a string whose character consist of the sequence of characters in this string beginning with position beginIndex and including those found up to, but not including, position endIndex. The variant that does not include endIndex includes characters from beginIndex to the end of this string.
- **String concat(String str)**: Returns a new string consisting of the characters in this string, followed by the characters in str.
- **String toLowerCase()**: Returns the characters in this string, with uppercase letters converted to their lowercase equivalents.

- **String toUpperCase()**: Returns the characters in this string, with lowercase letters converted to their uppercase equivalents.
- **static String valueOf(Object obj)** and overloads: Returns a new string containing a representation of the argument. For the variant with an Object parameter, returns "null" if obj is null, otherwise returns the value resulting from invoking obj.toString().

A.12.9 java.lang.Throwable

The class Throwable is the ancestor of iJava's exception class hierarchy.

```
public class Throwable {
    public Throwable();
    public Throwable(String msg);
        public String getMessage();
}
```

- **Throwable()**: Initializes a Throwable instance with a null message.
- **Throwable(String msg)**: Initializes a Throwable instance with the given message.
- **String getMessage()**: Returns the message with which this instance was initialized.

The subclasses of Throwable are as follows:

- **Error** represents exceptions triggered due to conditions such as out of memory or stack overflow.
- **Exception** represents exceptions that represent standard error conditions that may occur in iJava programs.
- **RuntimeException**, which inherits from **Exception**, represents standard error conditions which may be thrown by iJava throw statements.

The subclasses of RuntimeException are as follows:

- ClassCastException
- IndexOutOfBoundsException
- NegativeArraySizeException

Appendix B iJava Grammar

The following grammar for iJava has been adapted from the grammar presented in *The Java Language Specification, 3rd Edition* [19].

```
Identifier:
  IDENTIFIER

QualifiedIdentifier:
  Identifier { . Identifier }

Literal:
  IntegerLiteral
  FloatingPointLiteral
  CharacterLiteral
  StringLiteral
  BooleanLiteral
  NullLiteral

Expression:
  Expression1 [AssignmentOperator Expression1]

AssignmentOperator:
  =
  +=
  -=
  *=
  /=
  &=
  |=
  ^=
  %=
  <<=
  >>=
  >>>=

Type:
  Identifier { . Identifier } [TypeArguments] {[]}
  BasicType

TypeArguments:
  < Type {, Type} >

StatementExpression:
  Expression

ConstantExpression:
  Expression

Expression1:
  Expression2 [Expression1Rest]
```

```
Expression1Rest:
  ? Expression : Expression1

Expression2 :
  Expression3 [Expression2Rest]

Expression2Rest:
  {InfixOp Expression3}
  Expression3 instanceof Type

InfixOp:
  ||
  &&
  |
  ^
  &
  ==
  !=
  <
  >
  <=
  >=
  <<
  >>
  >>>
  +
  -
  *
  /
  %

Expression3:
  PrefixOp Expression3
  ( Expression | Type ) Expression3
  Primary {Selector}

Primary:
  ParExpression
  this [Arguments]
  super SuperSuffix
  Literal
  new Creator
  Identifier {. Identifier}[IdentifierSuffix]

IdentifierSuffix:
  [ Expression ]
  Arguments


NonWildcardTypeArguments:
  < TypeList >

PrefixOp:
  !
  ~
```

```
  +
  -

Selector:
  . Identifier [Arguments]
  [ Expression ]

SuperSuffix:
  Arguments
  . Identifier [Arguments]

BasicType:
  byte
  short
  char
  int
  long
  float
  double
  boolean

Arguments:
  ( [Expression { , Expression }] )

Creator:
  CreatedName ( ArrayCreatorRest | ClassCreatorRest )

CreatedName:
  Identifier {. Identifier} [NonWildcardTypeArguments]

ArrayCreatorRest:
  [ ( ] {[]} ArrayInitializer | Expression ] {[ Expression ]} {[]} )

ClassCreatorRest:
  Arguments

ArrayInitializer:
  { [VariableInitializer {, VariableInitializer} [,]] }

VariableInitializer:
  ArrayInitializer
  Expression

ParExpression:
  ( Expression )

Block:
  { BlockStatements }

BlockStatements:
  { BlockStatement }

BlockStatement :
  LocalVariableDeclarationStatement
  Statement
```

```
LocalVariableDeclarationStatement:
  Type VariableDeclarators ;

Statement:
  Block
  LeftHandSide AssignmentOperator Expression ;
  if ParExpression Statement [else Statement]
  while ParExpression Statement
  try Block Catches
  return [Expression] ;
  throw Expression ;
  break
  continue
  ;
  StatementExpression ;

Catches:
  CatchClause {CatchClause}

CatchClause:
  catch ( FormalParameter ) Block

MoreStatementExpressions:
  { , StatementExpression }

Modifier:
  public
  protected
  private
  static
  abstract
  final

VariableDeclarators:
  VariableDeclarator { , VariableDeclarator }

VariableDeclaratorsRest:
  VariableDeclaratorRest { , VariableDeclarator }

VariableDeclarator:
  Identifier VariableDeclaratorRest

VariableDeclaratorRest:
  {[]} [ = VariableInitializer]

VariableDeclaratorId:
  Identifier {[]}

CompilationUnit:
  [package QualifiedIdentifier ; ] {ImportDeclaration} {TypeDeclaration}

ImportDeclaration:
  import Identifier { . Identifier } [ . * ] ;
```

```
TypeDeclaration:
  ClassOrInterfaceDeclaration

ClassOrInterfaceDeclaration:
  {Modifier} (ClassDeclaration | InterfaceDeclaration)

ClassDeclaration:
  class Identifier [TypeParameters] [extends Type] [implements TypeList] ClassBody

TypeParameters:
  < TypeParameter {, TypeParameter} >

TypeParameter:
  Identifier

InterfaceDeclaration:
  interface Identifier [TypeParameters] [extends TypeList] InterfaceBody

TypeList:
  Type { , Type}

ClassBody:
  { {ClassBodyDeclaration} }

InterfaceBody:
  { {InterfaceBodyDeclaration} }

ClassBodyDeclaration:
  ;
  {Modifier} MemberDecl

MemberDecl:
  MethodOrFieldDecl
  void Identifier VoidMethodDeclaratorRest
  Identifier ConstructorDeclaratorRest

MethodOrFieldDecl:
  Type Identifier MethodOrFieldRest

MethodOrFieldRest:
  VariableDeclaratorRest
  MethodDeclaratorRest

InterfaceBodyDeclaration:
  ;
  {Modifier} InterfaceMemberDecl

InterfaceMemberDecl:
  InterfaceMethodOrFieldDecl
  void Identifier VoidInterfaceMethodDeclaratorRest

InterfaceMethodOrFieldDecl:
  Type Identifier InterfaceMethodOrFieldRest

InterfaceMethodOrFieldRest:
```

```
  InterfaceMethodDeclaratorRest

MethodDeclaratorRest:
  FormalParameters ( MethodBody | ; )

VoidMethodDeclaratorRest:
  FormalParameters ( MethodBody | ; )

InterfaceMethodDeclaratorRest:
  FormalParameters ;

VoidInterfaceMethodDeclaratorRest:
  FormalParameters ;

ConstructorDeclaratorRest:
  FormalParameters MethodBody

QualifiedIdentifierList:
  QualifiedIdentifier { , QualifiedIdentifier}

FormalParameters:
  ( [FormalParameterDecls] )

FormalParameterDecls:
  Type VariableDeclaratorId [ , FormalParameterDecls]

MethodBody:
  Block
```

Bibliography

[1] Paul F. Albrecht, Philip E. Garrison, Susan L. Graham, Robert H. Hyerle, Patricia Ip, and Bernd Krieg-Brückner. Source-to-source translation: Ada to Pascal and Pascal to Ada. *SIGPLAN Not.*, 15(11):183–193, November 1980.

[2] Apache commons math. `http://commons.apache.org/proper/commons-math/`, [Online; accessed 15-January-2016].

[3] Pablo Arrighi, Johan Girard, Miguel Lezama, and Kévin Mazet. The GOOL system: A lightweight object-oriented programming language translator. In *Proceedings of the 9th International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems PLE*, ICOOOLPS '14, pages 5:1–5:7, New York, NY, USA, 2014. ACM.

[4] Jan A Bergstra, Jan Heering, and Paul Klint. *Algebraic specification*. ACM, 1989.

[5] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Softw. Pract. Exper.*, 18(9):807–820, September 1988.

[6] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. a language and toolset for program transformation. *Science of Computer Programming*, 72(1):52–70, 2008.

[7] Btrace. `kenai.com/projects/btrace`, [Online; accessed 15-January-2016].

[8] Diego Ordonez Camacho, Kim Mens, Mark Van Den Brand, and Jurgen Vinju. Automated derivation of translators from annotated grammars. *Electronic Notes in Theoretical Computer Science*, 164(2):121–137, 2006.

[9] clang: a C language family frontend for LLVM. `clang.llvm.org`, [Online; accessed 3-April-2015].

[10] James R Cordy. Txl—a language for programming language tools and applications. *Electronic notes in theoretical computer science*, 110:3–31, 2004.

[11] Doxygen. `www.doxygen.org`, [Online; accessed 15-January-2016].

[12] Roger DuWayne II Duffey. Formalizing the expertise of the assembly language programmer. Technical Report WP-203, MIT Artificial Intelligence Laboratory, 1980.

[13] ECMA International. *Standard ECMA-262 - ECMAScript Language Specification*. ECMA International, 5.1 edition, [Online; accessed 3-April-2015].

[14] Mohammad El-Ramly, Rihab Eltayeb, and Hisham A Alla. An experiment in automatic conversion of legacy Java programs to C#. In *AICCSA*, pages 1037–1045, 2006.

[15] Emscripten 1.29.12 documentation. http://kripken.github.io/emscripten-site/, [Online; accessed 3-April-2015].

[16] Rodney Farrow and Daniel Yellin. Translating between programming languages using a canonical representation and attribute grammar inversion. Technical Report CUCS-247-86, Columbia University Computer Science Technical Reports, 1986.

[17] Gregory Gerard Faust. Semiautomatic translation of COBOL into HIBOL. Technical report, Massachusetts Institute of Technology, 1981.

[18] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[19] J Gosling, B Joy, G Steele, and G Bracha. *The Java Language Specification*. Addison-Wesley, 3 edition, 2005.

[20] J Gosling, B Joy, G Steele, G Bracha, and A Buckley. *The Java Language Specification Java SE 8 Edition*. Oracle, 8 edition, 2015.

[21] Google web toolkit. http://www.gwtproject.org/, [Online; accessed 3-April-2015].

[22] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF—reference manual. *SIGPLAN Notices*, 24(11):43–75, November 1989.

[23] Zef Hemel, Lennart CL Kats, Danny M Groenewegen, and Eelco Visser. Code generation by model transformation: a case study in transformation modularity. *Software & Systems Modeling*, 9(3):375–402, 2010.

[24] R. D. Huijsman, J. van Katwijk, C. Pronk, and W. J. Toetenel. Translating Algol 60 programs into Ada. *Ada Lett.*, VII(5):42–50, September 1987.

[25] International Organization for Standardization. *ISO International Standard ISO/IEC 14882:2014(E) Programming Language C++*. ISO, 2014.

[26] Junit. http://junit.org/, [Online; accessed 15-January-2016].

[27] Justin Koser, Haakon Larsen, and Jeffrey A Vaughan. SML2Java: a source to source translator. In *Draft Proceedings of the Workshop on Declarative Programming in the Context of Object-Oriented Languages*, page 105. Citeseer, 2003.

[28] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, 2nd edition, 1999.

[29] The LLVM compiler infrastructure. http://llvm.org/, [Online; accessed 3-April-2015].

[30] T. J. McCabe. A complexity measure. *IEEE TSE*, 2(4):308–320, December 1976.

[31] Arie Middelkoop, Atze Dijkstra, and S Doaitse Swierstra. Visitor-based attribute grammars with side effect. *Electronic Notes in Theoretical Computer Science*, 264(5):47–69, 2011.

[32] Vincent D Moynihan and Peter JL Wallis. The design and implementation of a high-level language converter. *Software: Practice and Experience*, 21(4):391–400, 1991.

[33] Nathaniel Nystrom, Michael R Clarkson, and Andrew C Myers. Polyglot: An extensible compiler framework for Java. In *Proceedings of the 12th International Conference on Compiler Construction*, pages 138–152. Springer, 2003.

[34] Free Pascal: Reference guide. http://freepascal.org/docs-html/ref/ref.html, [Online; accessed 3-April-2015].

[35] David Plaisted. An abstract programming system. In Susan Shannon, editor, *Leading-Edge Computer Science*, pages 85–129. Nova Science Publishers, 2005.

[36] David A Plaisted. Source-to-source translation and software engineering. *Journal of Software Engineering and Applications*, 6:30–40, 2013.

[37] J Prosise. *Programming Microsoft .NET*. Microsoft Press, Redmond, 2002.

[38] Puremvc. http://puremvc.org, [Online; accessed 15-January-2016].

[39] metrics. https://pypi.python.org/pypi/metrics, [Online; accessed 15-January-2016].

[40] The Python language reference. https://docs.python.org/3.4/reference/index.html, [Online; accessed 3-April-2015].

[41] ROSE compiler framework. http://en.wikibooks.org/wiki/ROSE_Compiler_Framework, [Online; accessed 3-April-2015].

[42] Stephen Schaub and Brian A Malloy. Comprehensive analysis of c++ applications using the libclang api. In *International Society of Computers and Their Applications (ISCA)*, 2014.

[43] Hampton Smith, Heather Harton, David Frazier, Raghuveer Mohan, and Murali Sitaraman. *Formal Foundations of Reuse and Domain Engineering: 11th International Conference on Software Reuse, ICSR 2009, Falls Church, VA, USA, September 27-30, 2009. Proceedings*, chapter Generating Verified Java Components through RESOLVE, pages 11–20. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[44] James Stanier and Des Watson. Intermediate representations in imperative compilers: A survey. *ACM Comput. Surv.*, 45(3):26:1–26:27, July 2013.

[45] Stringtemplate. www.stringtemplate.org/, [Online; accessed 15-February-2016].

[46] Simplified wrapper and interface generator. http://www.swig.org/, [Online; accessed 3-April-2015].

[47] Andrey A Terekhov and Chris Verhoef. The realities of language conversions. *IEEE Software*, 17(6):111–124, 2000.

[48] TIOBE Software: Tiobe index. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, [Online; accessed 3-April-2015].

[49] Andrew Tolmach and Dino P Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 8(04):367–412, 1998.

[50] Marco Trudel, Carlo A Furia, Martin Nordio, Bertrand Meyer, and Manuel Oriol. C to OO translation: Beyond the easy stuff. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 19–28. IEEE, 2012.

[51] Marco Trudel, Manuel Oriol, Carlo A Furia, and Martin Nordio. Automated translation of Java source code to Eiffel. In *Objects, Models, Components, Patterns*, pages 20–35. Springer, 2011.

[52] Mark GJ van den Brand, Arie van Deursen, Jan Heering, HA De Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter A Olivier, Jeroen Scheerder, et al. The ASF+SDF meta-environment: A component-based language development environment. In *Compiler Construction*, pages 365–370. Springer, 2001.

[53] R.C. Waters. Program translation via abstraction and reimplementation. *IEEE Transactions on Software Engineering*, 14(8):1207–1228, 1988.

[54] M. Weiser, A. Demers, and C. Hauser. The portable common runtime approach to interoperability. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, SOSP '89, pages 114–122, New York, NY, USA, 1989. ACM.