Scheme2Java

A functionally-equivalent source-to-source Scheme to Java translator

by
Ryan North

An honours project submitted to
Professer D. Deugo
in partial fulfillment of
the requirements for the degree of
Bachelor of Computer Science
and as part of the course 95.495

School of Computer Science
Carleton University
Ottawa, Ontario, Canada

April 11th, 2003

Abstract

The problem of translation from one programming language to another is not an easy one. There are different approaches to achieving such a translation, including emulation, byte-code generation, and source translation. Source translation from Scheme to Java has not been attempted before. Here, an implementation of automatic source translation from functional Scheme to imperative Java is discussed. This approach uses a Scheme-atomic model for translation, allowing it to be modular and extensible, as well as easy-to-understand. The results of this project, while not supporting the entire set of Scheme commands, are good. Outputted Java code is functionally equivalent, if verbose.

Acknowledgements

I would like to gratefully acknowledge the help of Professor Deugo in guiding my efforts towards the completion of this project and thank him for pointing out that my translation paradigm was the “Factory” model.

The obliging help of Patrick Wisking and Amanda Shiga in editing this document was gratifying and appreciated.

I would also like to thank my mother, Anna, for knitting the most excellent socks that kept my feet warm during the programming of this assignment.

Rex T.’s help in keeping me focused was also appreciated.

Table Of Contents

Abstract ii
Acknowledgements iii
Table of Contents iv
List of Figures vi
List of Tables vii

1 Introduction **1**
1.1 Problem 4
1.2 Motivation 5
1.3 Goals 6
1.4 Expected Contributions 7
1.5 Outline 8

2 Background and Related Work **9**
2.1 Options in Scheme Translation 9

3 Approach **12**
3.1 Overall Survey 12

4 Architecture **18**
4.1 Class Diagram 19
4.2 Helpful Objects 19
4.2.1 The *SearchString* Wrapper 20
4.2.2 The *StringInt* Data Type 20
4.2.3 The *TranslationResult* Data Type 21
4.2.4 The *Variable* Data Type 21
4.3 Data Types 21
4.4 Error Handling 23
4.5 Base Class 25
4.6 Beginning the Translation 26
4.7 Translating the Source 27
4.8 Translating Specific Commands 29
4.8.1 *primitive* 29
4.8.2 *ifStatement* 29
4.8.3 *math* 30
4.8.4 *lambda* 31

5 Validation, Verification, and Results **33**
5.1 Examining the Output 33
5.2 Appraisal of Results 42

6 Conclusions **46**
6.1 Results Achieved 46
6.2 Future Work 47
6.2.1 A More Complete Command Set 47
6.2.2 A More Intelligent Post-Translation 47
6.2.3 More Context for Translation objects 48
6.2.4 Translation to C++ and Other Languages 48
6.2.5 Other Future Work 49

References **50**

A Competing Options for Scheme Translation **51**

B Annotated Conversions **53**
B.1 Conversion 1 54
B.2 Conversion 2 55
B.3 Conversion 3 57
B.4 Conversion 4 59
B.5 Conversion 5 61
B.6 Conversion 6 62

C Instructions for Running the Software **64**

D Software and Documentation on Compact Disk (attached)

List of Figures

Figure 1.1: Adding The First Ten Integers in Java 3
Figure 1.2: Adding The First Ten Integers in Scheme 4
Figure 3.1: The Abstract Design Of The Framework's Translate Method 13
Figure 3.2: Schematic Translation of "(+ 2 (- 3 4) (* 4 5))" 14
Figure 4.1: Class Diagram for Scheme2Java 19
Figure 4.2: Selected listing of *SearchString* functions 20
Figure 4.3.1: Translated Scheme Code, Casting As Required 22
Figure 4.3.2: Translated Scheme Code, Using Functions 23
Figure 4.7: Listing of Supported Commands 28

List of Tables

Table 5.1.1: Output of Assignment 1 # 1a 34
Table 5.1.2: Output of Assignment 1 # 1b 34-35
Table 5.1.3: Output of Assignment 1 # 2 35
Table 5.1.4: Output of Variation on Assignment 1 # 6 35-36
Table 5.1.5: Output of Segment from Assignment 2 # 3 36-37
Table 5.1.6: Raw Output and Line Counts of All Testing 37-42

Chapter 1

Introduction

All computer programs are written in one or more **programming languages**, used as a way of representing the low-level computer functions – functions such as "take the value in memory location 0h223, add it to the value in memory location 1h232, and store the result in memory location 0h220" – in a way that is convenient and easy for humans to understand.

Obviously, different programming languages have different features and qualities. Programming languages can be considered in terms of **levels of abstraction**, where a higher level of abstraction is closer to normal human language and a lower level is closer to actual circuit-level computer code. The lowest level of abstraction is actual machine code, which is represented in binary and looks like this: "10101010111011001 01". It is possible to write programs in machine code, but this requires a tremendous level of knowledge and is extremely difficult to do well, to debug, to maintain, and to understand.

One level up is assembly-level languages, which are equivalent to the memory-location example above and look like this: "mov eax, 0d000h". Assembly programming was typically employed because, with a competent programmer, the code could be extremely fast and optimized. Still, however, this requires a lot of knowledge to develop and understand.

Today most programming is not done in assembly languages, but in **high-level languages**. Languages like C, C++, C#, Java, and Scheme are examples of such higher-level programming languages. These languages allow more abstraction (either in terms of objects or functions) which in turn allows for programmers to develop software quickly, with more ease, and hopefully make

it easier to maintain and reuse. High-level languages allow programmers to think more in terms of ideas, rather than how to implement those ideas on a specific hardware platform.

However even at this high level, choosing a programming language for a task is not trivial. Different programming languages make different fundamental assumptions about both the problem and the way in which it will be solved. These assumptions are called the programming language's **paradigm**.

For example, C++ and Java have an **object-oriented** paradigm, where all the code in a program is defined in terms of objects interacting with other objects. Here, you might see a line of code like "myCar.steeringWheel.turnToAvoid(shoppingCart)". Here, there is a myCar object, which has as part of it a steeringWheel object, upon which the operation "turnToAvoid" can be executed. This operation has passed into it a shoppingCart object. Presumably, the car object will turn to avoid the shopping cart.

C has an **imperative** paradigm, where code to perform a similar function might look like "wheelPosition = updatePosition(-1.2)". Imperative programming is characterized by sequential operations ("do this and then do this") and data assignment, such as in the previous example, where the variable "wheelPosition" is updated by the result of the "updatePosition" function. In some ways, the imperative programming paradigm seems to be the most intuitive, as it corresponds to the vague impression that most people have that computers are just following a recipe, step by step by step. In the imperative model, the steps are explicitly defined.

Imperative and object-oriented paradigms are not the only models to be found, however. Different programming paradigms include **logic programming**, as found in Prolog, and **functional programming**, used in Scheme. There are

even more paradigms (and further being invented or discovered), but we will constrain ourselves here to these four main paradigms (object-oriented, imperative, logic, and functional).

It is important to note a distinction here: C++ and Java are also imperative languages, but with an object-oriented paradigm included. Scheme too can approximate an object-oriented paradigm, but is a functional language.

Logic programming is where the programmer defines a series of facts about the universe and then queries the program on how those facts relate. For instance, if I am a sibling to my brother, and my brother only has male siblings, it could be deduced that I am male. Using logic programming, programs can easily made to make such logical deductions from a series of facts or assertions.

Functional programming is a more mathematical approach to software development, one that emphasizes the evaluation of expressions rather than the execution of commands. The regular variable updates seen in imperative and object-oriented paradigms ("x = 10") are not allowed in a pure functional language. As way of an example, consider code to add together the first ten whole numbers[1]. In Java's imperative/object-oriented model, the code might look like this:

```
total = 0;
for (i=1; i<=10; ++i){
        total += i;
}
return total;
```

Figure 1.1: *Adding The First Ten Integers in Java*

[1] Example from http://www.cs.nott.ac.uk/~gmh//faq.html#functional-languages

In Scheme's functional model, equivalent code looks like this:

```
(define sum
  (lambda (from total)
    (if (= 0 from)
        total
        (sum (- from 1) (+ total from)))))

(sum 10 0)
```

Figure 1.2: *Adding The First Ten Integers in Scheme*

Notice the differences: in Java, the value of a variable is repeatedly updated, while in Scheme, a function calls itself again and again until finishing.

Often, code that can be expressed in the functional model can be expressed in the imperative or object-oriented model, and vice-versa. That is what this project is about.

Scheme2Java is an attempt to automate translation from the object-oriented and imperative model of Java to the functional model of Scheme. Given Scheme source code as input, the program outputs functionally equivalent (code that does the same thing and gives the same results) Java source.

1.1 Problem

Translation is not an easy process. Computer translators for human languages still provide often-humourous output because human languages, despite their well-defined grammar, often have quirks in use that are difficult to capture in a computer program.

The advantage programming languages have in the translation problem is that they are well-defined and consistent – there is such a thing as incorrect computer code that does not make sense, much more than in spoken languages. Knowing this, then, shouldn't translation between two programming languages be easy, even trivial?

What makes translation from Scheme to Java non-trivial (in fact, quite difficult) is the fact that they employ different paradigms. To continue the comparison to human languages: this translation can be thought of not as translating from one similar human language to another, such as moving from English to French, but rather as automatically translating American Sign Language to Russian: two languages that make different fundamental assumptions about the world.

The rationale for such a translation lies in the very paradigm difference that makes translation difficult. The problem with different programming paradigms is that translation between them is difficult. If development of a program is begun in Scheme and later the decision is made to move to Java, there is no way to convert the source code, besides having a developer familiar with both languages do the translation by hand. There are other options besides the source-code conversion I've chosen (wrapping, emulation, and byte-code generation are some of the competing options listed in **Appendix A**), but I have found no reference in my research to any attempts at source-level conversion.

1.2 Motivation

The motivation for my project includes not only the practical case of translation when the choice of which language to use has been changed (an admittedly infrequent occurrence), but also:

- illustrating the difference in the content and design of functionally-equivalent source between the two languages;
- as a proof of concept, proving that such things can be done;
- doing something productive that has not been done before;
- learning more about the workings of the two languages.

The advantages of source-level conversion are many, and include being able to view the source and compare differences in output, being able to take advantage of Java-specific language features that are not mapped in Scheme before compiling, etc. Other approaches in conversion attempt to emulate this advantage by extending the Scheme language with program-specific extensions, a much more limited approach.

Since the program will be compiled into bytecode and run by the native Java software, any optimisations that apply to a regular Java programs also apply to the converted-from-Scheme Java source.

1.3 Goals

The goal of the project is to have functionally-equivalent source conversion from Scheme to Java. Realistically, there is no way the complete Scheme specification, which runs fifty-pages long[2], could be handled in what is considered a one-semester project. My goal for this project is translation of the most-useful and frequently-used Scheme commands. The project uses a Scheme-atomic model for source conversion, wherein basic Scheme constructs are translated and used as building blocks in the translation of an entire program. I believe this approach will be used to generate valid output, but it may be that multiple passes will be necessary. Syntactic validation of the

[2] "Revised[5] Report on the Algorithmic Language Scheme", edited by Richard Kelsey, William Clinger, and Jonathan Rees , at http://www.schemers.org/Documents/Standards/R5RS/HTML/

outputted source is done using Java's own javac compiler, while functional validation is accomplished by actually comparing the output of the both versions (Scheme and Java) of a translated program, and by examining the source.

Knowing that there is no way that I can fully complete this project myself has implied some further goals on the project: the program I produce should be easy-to-understand, modular, object-oriented, stable, well-documented, and learning from the source be simple.

1.4 Expected Contributions

What follows is a list of contributions I expected would be necessary to achieve the goal of a functionally-equivalent source Scheme to Java translation, and my comments on those contributions:

- choosing the translation model

  A Scheme-atomic model was chosen, as discussed.
- translating the most useful and interesting Scheme commands

  A list of these might include define, all the basic math commands, write, quote, lambda, set!, and as many primitives as possible
- implementing these translations in a Java framework
- writing various helper methods commonly used in Scheme translation
- designing the Java implementation to be easy, extensible, and fast

  A factory paradigm for the basic framework was chosen to allow the code to be easy-to-understand and extensible.

The hardest part of this was expected to be the translation and the implementation of the translation.

1.5 Outline

The rest of this report is organized as follows:

Chapter 2 gives a background on Scheme to Java machine translation and reviews other relative work in the field.

Chapter 3 discusses the overall approach to the translation problem, proceeding from a high level survey of the approach down to the specifics of its various aspects.

Chapter 4 discusses the architecture of the implementation of the translation. This again takes the expository approach of going from the top-down. First a class diagram is presented, then details on helper objects, data types, and the philosophy taken towards error handling in the assignment. Then the structure of the translation aspect of the program is explored: first the abstract base class, then down to the Framework, finally arriving at the low level of the Translation-object atoms.

Chapter 5 shows how the results were verified, and includes a comparison of the output of various test cases in both their original Scheme form and their translated Java form.

Chapter 6 concludes the report by summarizing the contributions made, comparing the results with previous work, suggesting areas for future improvement, and by giving a frank assessment of the results obtained.

Chapter 2

Background and Related Work

In this chapter, previous attempts at Scheme to Java translation are reviewed, by way of a survey of the field, and commented upon. The competing efforts are reviewed in comparison to the translation approach I've chosen.

2.1 Options in Scheme Translation

Translation between Scheme and Java is not a new field, and has been accomplished before. However, my project is the first to translate source to source. Other translation projects have relied on two different techniques. These techniques are:

- running Scheme source in a Java-based Scheme emulation environment
- converting the Scheme source to directly to Java bytecode (the approach taken by the software package "Kawa" [3] alone)

The first option is uninteresting and don't really solve the problem of translation. Rather than actually converting the Scheme source, this solution just creates an environment where the Scheme source can run unmodified. This is not translation, but rather emulation. The Scheme source is not converted, but placed in an environment where it can run unaltered.

Kawa's conversion is more interesting. Rather than converting source-to-source, as I do, Kawa converts the scheme into internal "Expression" object, which it then compiles into Java bytecode. Java bytecode is the lowest form of Java code; it is similar to the machine-language ones and zeros discussed

before. The advantage of Java bytecode is that it is designed to be platform independent, to run on any machine for which a Java Virtual Machine has been developed. When one wants to run Java source code (like what my program outputs), they compile it into Java bytecode.

What this means is that Kawa straddles the boundaries between emulation and source-to-source conversion. The source is transformed, but in a way that it can be emulated by the Java virtual machine. There are advantages and disadvantages to this approach.

The advantages of directly-generated bytecode seem to me to be small: you can input Scheme code and have Java execution on-the-fly without resorting to the file system; but with this gain you lose the clarity of being able to see the generated Java source. While my batch translation would lose the interactivity of a live Scheme interface, any queries one might want could be added to the Scheme source before conversion or manually to the Java source, after conversion. The interactivity of a Scheme environment could also be emulated (unfortunately, with a performance hit) by invisibly regenerating the source and executing again.

The advantages of a translator, for my purposes, outweigh any losses: these advantages include being able to view the source and compare differences in output, being able to take advantage of Java-specific language features that are not mapped in Scheme before compiling, etc. Other approaches in conversion attempt to emulate this advantage by extending the Scheme language with program-specific extensions, a much more limited approach.

Since the program will be compiled into bytecode and run by the native Java software, any optimisations that apply to a regular Java programs also apply to

[3] Kawa can be found online at http://www.gnu.org/software/kawa/, and is described in Appendix A.

the converted-from-Scheme Java source, an advantage the generated-bytecode of Kawa lacks.

A selected listing of competing Scheme translation efforts is found in **Appendix A**.

Chapter 3

Approach

In this chapter the approach I've chosen to Scheme translation, at a high level, is explored and justified. This explains the "philosophy and science" behind the translation of Scheme code in the software.

3.1 Overall Survey

The translation approach, at a high level, involves translating individual scheme command atomically - that is to say, independently. So, the programming to translate "define" is separate, and in a different object, from the programming to translate "lambda".

The management of these translation atoms is handled the *Scheme2JavaFramework,* which roughly follows the "Factory" pattern[4]. This pattern allows the *Framework* to act like a dispatcher, identifying the Scheme code to be translated, creating the appropriate *Translation* object, and letting it do the work. When the object finishes, the *Framework* returns the result to the object that originally requested it.

The *Framework* is an object that can be created by a user interface, either a command-line interface (implemented as *Scheme2JavaCLI*) or as a GUI (unimplemented). Once created, one calls its method translate(). This method causes the *Framework* to enter a translation cycle, which, abstractly, is as follows:

[4] A nice definition of the Factor pattern and some exploration of its advantages can be found at http://gsraj.tripod.com/design/creational/factory/factory.html

while there is still input code left to translate
    select the next segment of bracketed code
    identify which Scheme command is contained in the segment
    create the appropriate Translation object for that command
    call that object's translate() function
end while

Figure 3.1*: The Abstract Design Of The Framework's Translate Method*

Translation is entirely recursive.

There are two kinds of translation objects. The first, the *Framework*, simply identifies the command to be translated, creates the appropriate object, and returns the result of that object's translation work. The second kind of translation object is that which is created by the first: the objects that do the actual translation. There are many examples of such objects: plus, minus, define, lambda, etc, but there is only one *Framework* object. This does not mean, however, that the *Framework* is static. Many *Framework*s may be created during the translation process. Consider:

Any created object may find another Scheme command within it – for example, (+ 2 3) could just as easily be (+ 2 (+ 4 5)). When such code is discovered, the object doing the translation recognizes that it doesn't know how to handle this code, and instead creates a new *Framework* object, giving it the segment to be translated. The Framework identifies the code, creates the proper object to translate it, and waits. Once the translation is complete, the resulting Scheme code, as well as some information about its properties, is returned by the *Framework* to the object that requested the translation initially. The object can then continue with its translation.

As an example, recursion for (+ 2 (- 3 4) (* 4 5)) would look, schematically, like this:

framework: translating "(+ 2 (- 3 4) (* 4 5))"
framework: this is the "plus" command, creating a plus object to translate this code.
plus translator: translating "(+ 2 (- 3 4) (* 4 5))"
plus translator: found another scheme command "(- 3 4) (* 4 5)", sending it to translate
framework2: translating "(- 3 4) (* 4 5)"
framework2: translating first part, "(- 3 4)"
framework2: this is a "minus" command, creating a minus object to translate this code.
minus translator: translating "(- 3 4)"
minus translator: returning result
framework2: got result for "(- 3 4)", now translating "(* 4 5)"
framework2: this is a "times" command, creating a times object to translate this code.
times translator: translating "(* 4 5)"
times translator: returning result
framework2: got result for both "(- 3 4) (* 4 5)", returning it to the plus translator that requested it originally
plus translator: got result, using it to finish my translation.
plus translator: returning result of "(+ 2 (- 3 4) (* 4 5))" to base translator that requested it originally
framework: got result, translation finished.

***Figure 3.2**: Schematic Translation of "(+ 2 (- 3 4) (* 4 5))"*

Notice how there are two instances of the *Framework*: the one that began translation on the entire chunk of code, and the one that was created by the plus translator to translate some code that it didn't know how to deal with.

As you can imagine, many *Framework* objects get created during the translation process. They all behave identically, identifying the code chunk, passing it off to be translated, and returning the result.

The reason there is not one static *Framework* handing all translations is that there is a lot of information that must be stored about the translation and its results: what variables were used, how much code was translated, which functions were created and which were referenced, etc. It is easier and more intuitive if there is one *Framework* handling the identification and translation dispatch for each unknown Scheme command.

The advantages of this "factory"-model system of the *Framework* and creating translations objects are:

- code is separated in a way that is intuitive and logical
- code that handles one element of Scheme translation (adding numbers) need not worry about other elements (defining functions)
- new translation objects can be added simply by creating the translation code and adding an identifying reference to the Framework, allowing flexibility and scalability
- any problems in code can be localized to the object they are occurring in, as a problem in lambda translation could only be the fault of the lambda translator.

The disadvantage is that code may be duplicated between translation objects, as many of these objects may perform similar tasks in the translation. However, this is minimized by having all translation objects (including the

Framework) inherent from a base abstract *Translator* class, which defines many useful functions common to all translations.

One wrinkle in the general translation process is that, before the first *Framework* translation method is called, the *Scheme2JavaCLI* (and presumably, the GUI and any other client of the *Framework*) invoke a preTranslate() method. After the translation, a postTranslate() method is similarly invoked.

The preTranslate() method does various transformations to the input code to make it easier to translate. Tabs and extra spacing are removed, comments are stripped off, and some unsupported Scheme commands (like "cond") are transformed into functionally-equivalent and supported scheme commands (like "if"). This allows the translation to handle more Scheme atoms than it could otherwise do. The preTranslate also does a basic syntax check on the Scheme source, ensuring that there are enough close brackets to match all the open brackets.

The postTranslate() method performs some re-formatting (proper tabulation, semicolons, etc) to make the code syntactically correct, easier-to-read, and presentable. It also applies the outer shell of the Java program, code which supports the translated scheme source.

Notice how the *Framework*, in effect, generates a translation tree on-the-fly. I had considered first creating a translation tree, representing the Scheme code, but the disadvantage to this is more overhead and rigidity. The *Framework* follows a tree-recursion down through the Scheme code, but does not formalize it in a data structure, allowing different approaches to be taken. For instance, a specific Scheme translation object could, upon returning, tell the Framework

to go back or forward an arbitrary number of characters. While this functionality wasn't used in this project, it's nice to have there.

This is the general approach to translation. What follows are specifics in the implementation.

Chapter 4

Architecture

This chapter explores the lower level architecture for the program. Details such as a class diagram, helper objects, data types, and error handling are discussed. An exploration of translation, along with examples of translating specific commands, is presented.

4.1 Class Diagram

As the actual translation of Scheme commands is done by specialized objects, a hierarchy is needed in order to use code efficiently. What follows (on a separate page, due to its size) is a class diagram for Scheme2Java:

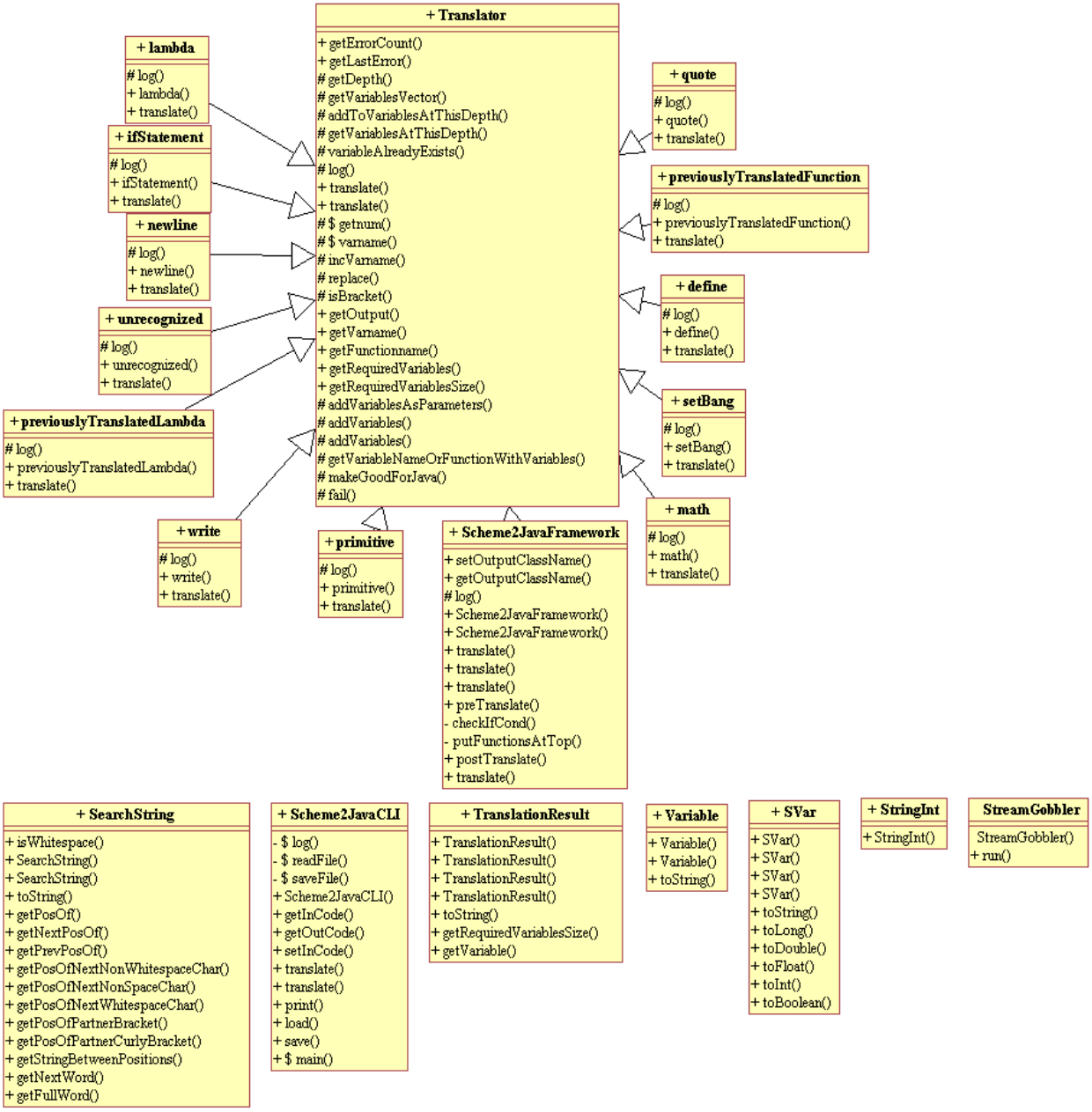

***Figure 4.1**: Class Diagram for* Scheme2Java

4.2 Helpful Objects

In the development of the structure of Scheme2Java, various objects were designed to make the translation process easier. What follows is a description of some of them and their justification.

4.2.1 The *SearchString* Wrapper

Early in the development process, it because obvious that most of the translation would be done with manipulations to *String* objects. The plan was to make a new subclass of *String*, one that would have various Scheme-specific string-manipulation functions included. Unfortunately, for reasons that are unclear, Java defines *String* as a final class – no subclasses are allowed.

For this reason, *SearchString* is a transparent wrapper class for *String*. It contains a public-accessible *String*, called str, as well as several public functions that allow manipulations on the *String* str. These functions are low-level, straightforward, and are for the most part uninteresting, but a brief listing should give you a flavour of the type of things that are possible with *SearchString*s:

```
public int getPosOf(String toFind);
public int getPosOfNextWhitespaceChar(int start);
public int getPosOfNextNonWhitespaceChar(int start);
public int getPosOfPartnerBracket(int start);
public String getStringBetweenPositions(int start, int finish);
public StringInt getNextWord(int start);
```

***Figure 4.2**: Selected listing of* SearchString *functions*

4.2.2 The *StringInt* Data Type

StringInt is designed as a basic class containing a public *String* and *int*, used in many of the string manipulation functions in *SearchString* to return both the result of the manipulation, as a *String*, and the position of that manipulation, as an *int*.

4.2.3 The *TranslationResult* Data Type

TranslationResult is used to return the result of a translation. It contains *String*s representing the name of the returned variable (or function, if one is created in the translation process), the code, a *Vector* of the variables the code requires, and a boolean indicating whether or not the translated code represents a lambda.

4.2.4 The *Variable* Data Type

Variables is used to store the result of a variable binding. It holds two strings, a name and a value. Variables are stored in a static Vector in the *Translator* object, called "variables", allowing any Translation object to see the recorded state of the Scheme environment, and modify it, at will. There are helper methods in *Translator* for this.

4.3 Data Types

One difficulty in translating Scheme code to Java is variable typing. Java has strict typing, requiring each variable to specifically announce what type of information it represents. Scheme on the other hand is more lose, allowing one to define a number as, say "1701", and not having to worry whether or not that number is an integer, long, float, or any other sort of numbered data type.

Translating from a loosely typed language to one with strict typing therefore presents a challenge. How does one determine what type of data "1701" is? One option is to look at all the ways in which the number is used, and from that deduce the data type that is required.

The downside with this option is that you must, in essence, interpret the Scheme code before you translate it, running through its execution to figure out how the number is used. Not only is this costly, it's difficult to do, and the Scheme code may be executed in a way your interpreter does not predict. This option was therefore rejected.

The option I settled upon was to, in effect, avoid this problem of determining variable type altogether by not deciding what type a variable is until the question becomes pertinent. That is, variables would be defined as *String*s in the Java source code, the most malleable and castable data type available. Only when the code required numbers to be added or divided would I convert, on the fly, to a numbered data type.

Code would look like this:

```
String var1 = "2";
String var2 = "3";
String var3 = (Integer)var1.intValue() + (Integer)var2.intValue();
```

Figure 4.3.1*: Translated Scheme Code, Casting As Required*

Notice how the variables are cast on-the-fly to what is required.

The downside is that, besides making the code ugly, it required a lot of nested casts when translating equals. Plus, you were limited to what you could cast *String* to, and it made the code exception-prone. As a solution, I decided that all variables would be cast to SVars (short for "Scheme Variables"), a data type containing a *String* and including methods to cast to doubles, which are the numeric data type with the most precision. Code now looks like

```
SVar var1 = new SVar("2");
SVar var2 = new SVar("3");
```

```
SVar var3 = new SVar(var1.toDouble() + var2.toDouble());
```

Figure 4.3.2: *Translated Scheme Code, Using Functions*

The downside to this approach is that a toDouble() method must be included in the SVar, but this is not too bad. I had considered writing add(), subtract(), multiply(), etc methods to the SVar, but the downside to this is that a method must be written for every combination of numbers: add(x), add(x, y), add(x, y, z), etc, to infinity. This would result in somewhat clearer code, but the cost of such a tradeoff was too high, even if the SVar object, included in all translated Scheme output, can have extra methods added or removed easily.

This is still ugly code though. I had considered using add()-style functions when the program knew that they were supported for that number of arguments, and falling back to the toDouble()-style when they weren't, but this would be two different styles for no functional reason. The two do behave slightly differently, and this could end up being trouble down the road. For this reason I stuck with toDouble()-style casting.

Had I chosen to convert the Scheme source to C++, operator overloading would have been a good option to use with the Svars. I could have had code like "SVar res = var1 + var2;", where var1 and var2 are both Svars. Alas, operator overloading is disallowed in Java. Hindsight is 20/20.

4.4 Error Handling

All of the functions in a *SearchString* (and indeed, in the Scheme2Java program) are designed to return null if there is an error in input, or the function cannot do what is requested of it. This was an architectural decision, and its motivation is as follows.

Returning a blank string (“”) may be indistinguishable from correct output, so that is not a good idea. Throwing a user exception would allow one to provide more details on the error and the situation in which it occurred, but this, to be useful, requires exception handling at all levels of the application. Since *SearchString*s are used throughout Scheme2Java, it would be a bad idea to have every Translator subclass know how to handle an OutOfBound exception, as distinct from an InvalidInput exception, different from an OutOfInput exception. The other option is to handle exceptions at the base level of the program – which is in fact what happens.

Returning null when something else is expected will cause an exception – a NullPointerException. The base class handles this exception, printing out what happened and the circumstances in which it happened. This, combined with the logging that is written to the file and the screen, is enough to determine the place and circumstances in which the error occurred. So rather than worrying and creating specific Exception classes for each error that can occur, we return “null” and trust that the exception that causes will be enough to determine where the error took place.

A final advantage with returning null to indicate errors is that it is easily checked for and coded in Java, when compared with the more awkward try/catch structure that is required for exceptions. This allows Translators to work with the nulls they expect, using them to indicate when, for instance, the end of the input code has been reached.

Of course, this does have the disadvantage of flattening the error structure, so that an “end of input” null is indistinguishable from a “error in input” null – but if there is an error in input the translation will not succeed anyway, so this disadvantage is acceptable.

In the case of serious errors, where there is no way to gracefully recover and the translation has no chance of succeeding, the program exits with an error message at the point when the fault is discovered, and emphasizes that it is terminating irregularly.

4.5 Base Class

The base class is an abstract *Translator* object. This class contains methods and properties that are common to all objects that will be doing Scheme to Java translation. It is abstract, which means that it cannot be created directly, but other object can inherent from it. This allows it to define methods that must be included in all translation objects, controlling their design. It also allows it to provide useful methods and properties that anyone translating would find useful. For instance, all translators need to keep track of:

- their input code
  - this is stored as *SearchString* inCode
- their output code
  - this is stored as *String* outCode
- what variables they've used, and what functions and lambdas they've referenced
  - these are stored as *Vectors* requiredVariables, functionList, and lambdaList
- the Scheme environment
  - this is stored as the static *Vector* variables

Also, all translators need to know how to

- translate source code (obviously)
  - -the abstract method translate defines this
- log their actions to the screen and log file
  - -the method log(String s) does this. Child objects are expected to

override this function with one that prepends their name to the string being logged, so “created” becomes “define: created.”

There also some helper functions, like

- a function to format their list of variables as a Java function signature
- etc.

This abstract Translator base class does include one private static variable, a number simply called “num”, that can only be accessed through protected methods. “num” is used to name variables and functions in a way that is unique. Each access of the value of “num” increases it by one, so that when a translator wants to name a variable, it need only place getnum() as a variable name to be ensured of uniqueness. Additionally, it holds a static “errorCount” number and “lastError” string, used for counting and keeping track of non-critical errors that may be encountered in the translation process. This isn’t used much in the present software (few translation errors are non-critical), but one example of its use occurs when the depth of a variable in the Scheme code cannot be determined, for whatever reason.

4.6 Beginning the Translation

Translation is begun using *Scheme2JavaCLI*, the command-line interface (CLI) for the Scheme2Java Translator. It is a wrapper for the *Scheme2JavaTranslator* object, allowing easy access to translation.

When invoked at the command line, it retrieves and feeds the *Scheme2JavaTranslator* object input code, runs its preTranslate() method, then its translate() method, and finally its postTranslate() method. It then saves the output to the specified file.

The *Scheme2JavaTranslator* is encapsulated in this way to allow multiple uses of the same translation functionality. This access could be done through the CLI (as implemented), invisibly as part of a larger program, or through a GUI.

Scheme2JavaCLI is used at the command-line as follows:
Scheme2JavaCLI [**inputfile**] [**classname**] [**logging level**] **[javac path]**

Inputfile is the filename of the Scheme source one wants translated.
Classname is the class name for the outputted Java source.
Logginglevel is either '1', '2', or '3', and controls the amount of data that is written to the console as the translation proceeds. For most translation '2' is all that is required to get a feel of the recursion going on, but to see everything that is happening, '3' is required.

If options are specified incorrectly, the correct usage is printed to the console and the program exits. There is more detailed information on usage in **Appendix C**.

4.7 Translating the Source

The *Scheme2JavaTranslator,* as specified before, has methods to preTranslate, translate() and postTranslate().

The translate() method uses *SearchString*s to determine what the first command in the passed Scheme is. It then creates the appropriate translator, passes it off the code, and waits for it to return. If there are more commands left to translate, it translates them similarly.

This becomes recursive, as the created *Translator* object may well encounter further Scheme code to be translated. This is passed off to a new *Scheme2JavaTranslator* object, as explained above.

The *Scheme2JavaTranslator* currently recognizes the following commands:

/
+
-
*
=
<
>
<=
>=
modulo
abs
round
and
or
not
remainder
write
quote
newline
if
lambda
define
set!

Figure 4.7: *Listing of Supported Commands*

…as well as any other previously-translated functions. For instance, if you define a function “divide-by-two”, code like “(divide-by-two 10)” will be translated properly.

If the command is not recognized, the Translator will create an *unrecognized* object, which does a best-case translation of simply assigning the code, untranslated, to a variable. The reason it does this instead of returning an error is that this is more optimistic, as it sometimes works for translating Scheme constructs that haven’t been specifically programmed for. Numbers were originally translated this way, but a *primitive* object was created for them. The two classes (*unrecognized* and *primitive*) do essentially the same translation, but *primitive* translates #t and #f correctly.

4.8 Translating Specific Commands

Details on selected *Translator* classes follow. Not all are included, but those with interesting features or properties have those explored.

4.8.1 *primitive*

The class *primitive* is the simplest translation possible. *primitive* is used when the Scheme code to be translated is one of the base units: usually, a number. For instance (+ 2 3) gets broken down into +, 2, and 3. 2 and 3 get translated by this code, which simply puts the numbers into SVars and returns them. 2 gets translated into SVar var1 = new SVar(“2”);

4.8.2 *ifStatement*

A scheme “if” structure has 3 parts: the test, the consequent, and the alternate. The test is what controls execution through the if block, the

consequent is what happens if the test results in a true statement, and the optional alternate is what should happen otherwise.

To translate this, the *ifStatement* object parses out the scheme code for the test, consequent, and if it is there, the alternate. It then sends these chunks of Scheme code to be translated (using a new *Scheme2JavaTranslator*) and formats the results in a simple Java "if" block.

The code for this is simple, and that's why this class was chosen as the first example. All of the real translation is done recursively by whatever objects are responsible for the code contained in the Scheme chunks. i*fStatement* shows the power of translating recursively: now matter what test is done in the if statement, all the *ifStatement* object has to worry about is putting the result in the proper Java structure.

4.8.3 *math*

math is used very often, as most Scheme code (indeed, most computer code) has some sort of high-level numerical aspect to it. *math* handles basic mathematical operations such as +, -, etc. To do this, it first identifies what the operation of the code is. If the operation is one that requires slightly different formatting, this is noted. For instance, abs, not, or and or round all require the operation to be a function in Java ("Math.round(x)") while other operations like + or * only require that they be put between operands ("x + y").

Four vectors are then created. One stores the list of variables, one the code, one the list of required variables, and one with a list of functions. These vectors are required for recursion: if the Scheme code contains sub-chunks that require translation, the outer chunks must know what variables need to be passed in. These variables, function names, and code listings are stored in

these Vectors until they are required. For example, a variable might not be used until 5 or 6 layers deep in the Scheme code: in the resulting Java code this variable must be passed along to the correct function. This will become more clear with an example; if you wish, flip to **Appendix B** now to see how this takes place.

Once the recursion is complete, the top-level of the code is transformed. This transformation is simple, but does depend on the results of the recursion. Depending on the complexity of the code, the recursion may have returned either a variable or a function. In the variable case, the transformation turns "(+ x y)" into "x.add(y)". In the function case, the transformation turns "(+ x (+ 2 3))" to "x.add(function1())", where function1 adds 2 to 3. Here we use the Vectors we populated during the recursion to ensure that the output code is correct and contains everything we need it to.

Again, like *ifStatement*, the actual code is pretty straightforward. About half the code handles the recursion, the other half handles the transformation of the results of the recursion.

4.8.4 *lambda*

lambda is a special case because it is a more difficult translation. lambdas in Scheme represent a function, and there is no simple mapping from a function in Scheme to some construct in Java. I had several options:

• I could have implemented lambdas as separate classes. The downside to this is that there may be a lot of classes, plus, this is a lot of overhead. It also makes passing lambdas into functions more difficult. If I had elected to translated into C++ instead of Java, this would be easier, as there are ways to pass functions as arguments in C++. It is not so in Java.

• I could have implemented lambdas as anonymous classes. This would have been good and I was leaning towards this option for a while, but the downside is that handling recursion in the Scheme code would be very difficult, plus, there are many restrictions on anonymous classes in Java.

• I could have pre-translated the Scheme input to remove lambdas and replace them with named expressions. This would solve the problem but would likely have been difficult, especially with regards to recursion. Besides, this is a cheap solution in that it solves the interesting problem of lambda translation by removing it completely. This would have been my last-resort option, had I not figured out how to implement lambdas as named classes.

To implement lambdas as named classes, each lambda would be represented as a class – in the same source file - with various methods and arguments. The downside to this technique is that defining the form for the class appeared to be difficult. I decided on having an execute(a, b, c, … n) method. Each class contains the variables it requires, enough to represent the environment it was created it. The downside to this is that I had to code for recognizing lambda-objects elsewhere. An almost-equivalent option was to turn the lambdas into objects with the same methods and attributes. There was no real advantage to classes over objects, but I preferred the class translation as a matter of personal taste.

Chapter 5

Validation, Verification, and Results

This chapter explores how the output from the program was validated and verified, and also presents a frank appraisal of its output.

5.1 Examining the Output

The output of the program was verified using Java's own compiler, Dr. Scheme, and my own knowledge of both languages. The Scheme interpreter used was Dr. Scheme version 103p1, the Java compiler and interpreter was version 1.4.1_01. The java compiler checked that the output code was technically accurate; I checked that it was functionally-equivalent. To do this I compared the output of the two programs to make sure that they were identical. Identical output suggests that the programs are functionally-equivalent, as the same input results in the same output, but there is a chance that this could be a fluke. Examination, by hand, of the Java source code was necessary to convince myself that the two versions of the code would behave the same in all circumstances. This requires some knowledge of Java to be able to judge whether or not the output of the Scheme2Java translator is functionally-equivalent, but there is no way to automate this. Were this software released publicly, users would either have to take the leap of faith that the code is functionally-equivalent (they would have to trust the program and its developer(s)), or look at the outputted source themselves. After each translation object was completed, it was robustly tested to ensure that it operated as expected and handled errors gracefully.

Here is a table of various translations attempted (more detailed examples are included in **Appendix B**, all of them are included on the attached CD in

Appendix D) and the output from both versions (Scheme and Java) of the source code. Along with the comparison of outputs, notice the line counts for both versions of the code. This will be explored in section 5.2.

Scheme Code (95.307 Assignment 1 # 1a):

```
(define (f a b)
        (/(+ (modulo a 2) 5)  (* 9 (abs b)))
)
(write (f 10 11))
```

Scheme Line Count	Java Line Count	Scheme Output	Java Output	Remarks
4	46	5/99 [= 0.05050…]	0.05050…	Java doesn't keep track of fractions like Scheme does, so the result is automatically put into decimal form.

Table 5.1.1: Output of Assignment 1 # 1a

Scheme Code (95.307 Assignment 1 # 1b):

```
(define (g a b c d x y )
        (and (or (= a b) c) (or (and d (not c)) (> x y)))
)
(write (g 1 2 #f #f 10 20))
```

Scheme Line Count	Java Line Count	Scheme Output	Java Output	Remarks

4	61	#f	false	This example uses a valid test case, but if another one were used that didn't make sense (for instance, (g 1 2 3 4 5 6)) the Scheme code wouldn't run and the Java code would give incorrect output. Garbage in, garbage out.

Table 5.1.2 – Output of Assignment 1 # 1b

Scheme Code (95.307 Assignment 1 # 2):

```
(define (inv x)
        (/ 1 x)
)
(define (totalResistence r1 r2)
        (inv(+ (inv r1) (inv r2) ))
)
(write (totalResistence 10 100))
```

Scheme Line Count	Java Line Count	Scheme Output	Java Output	Remarks
7	37	100/11 [=9.090909…]	9.09090909…	The Java code also creates functions called "totalResistence" and "inv" in the translation, and calls them by name.

Table 5.1.3 – Output of Assignment 1 # 2

Scheme Code (variation on 95.307 Assignment 1 # 6):

```
(define (allSame x y z)
        (and (and (= x y) (= x z)) (= y z))
)
(define (notAllSame x y z)
```

```
        (not (allSame x y z))
)
(write (allSame 10 10 10))
(write (allDifferent 10 10 10))
```

Scheme Line Count	Java Line Count	Scheme Output	Java Output	Remarks
8	64	true false	true false	Like the Scheme code, the Java code defines allDifferent as !(allSame).

Table 5.1.4 – Output of Variation on Assignment 1 # 6

Scheme Code (part of 95.307 Assignment 2 # 3):

```
(define (len num)
        ;returns the number of digits in the non-negative integer, 11 digits or
less
        (if (< num 10)
                1
                (if (< num 100)
                        2
                        (if (< num 1000)
                                3
                                (if (< num 10000)
                                        4
                                        (if (< num 100000)
                                                5
                                                (if (< num 1000000)

                                                        6
                                                        (if (< num 10000000)
                                                                7
                                                        (if (< num 100000000)
                                                                8
                                                        (if (< num 1000000000)
                                                                9
                                                        (if (< num 10000000000)
                                                                10
                                                                11
                                                        )
                                                        )
```

)))))))) (write (len 1000))				
Scheme Line Count	**Java Line Count**	**Scheme Output**	**Java Output**	**Remarks**
35	151	4	4	The more verbose the Scheme code, the more verbose (comparatively) the Java code is.

Table 5.1.5 – Output of Segment from Assignment 2 # 3

Here is a table showing all the example cases on the attached CD, and comparing their output and line sizes pre- and post-translation. Results are highlighted. For more information on the examples, see the \Examples directory on the CD. Line counts are slightly inflated here because testing is included – but again, these line counts should be taken as only a rough measurement of efficiency: any Scheme program could be written on one line, and quality of output cannot be measured by line counts alone. These line counts are for standard formatting in both languages, unless otherwise noted.

Filename	**Scheme Line Count**	**Java Line Count**	**Scheme Output**	**Java Output**
AssignmentOneNumberOneA	25	73	"Test case 1: Testing 0 1 – expect 5/9" **5/9** "Test case 2: Testing 1 1 – expect 2/3"	Test case 1: Testing 0 1 - expect 5/9 **0.555555555 5555556** Test case 2: Testing 1 1 -

			2/3 "Test case 3: Testing 9999 99999 – expect 2/299997" **2/299997** "Test case 4: Testing 0 0 - expect division by 0" **/: division by zero**	expect 2/3 **0.666666666 6666666** Test case 3: Testing 9999 99999 - expect 2/299997 **6.666733334 000007E-6** Test case 4: Testing 0 0 - expect division by 0 **Infinity**
AssignmentOneNumberOneB	22	88	"Test case 1: Testing 0 0 #f #f 0 0 - expect f" **#f** "Test case 2: Testing 1 1 #f #t 99 22 - expect t" **#t** "Test case 3: Testing bad input 4 5 2 3 #t #f - expect error" **Condition value is neither true nor false: 2**	Test case 1: Testing 0 0 false false 0 0 - expect f **false** Test case 2: Testing 1 1 false true 99 22 - expect t **true** Test case 3: Testing bad input 4 5 2 3 true false - expect error **false**
AssignmentOneNumberTwo	32	64	"Test case 1: Testing 10 10 - expect 5" **5** "Test case 2: Testing 1 1 - expect 1/2" **1/2** "Test case 3: Testing -1 -1 - expect -1/2" **-1/2** "Test case 4: Testing 0 0 - expect division	Test case 1: Testing 10 10 - expect 5 **5.0** Test case 2: Testing 1 1 - expect 1/2 **0.5** Test case 3: Testing -1 -1 - expect -1/2 **-0.5** Test case 4: Testing 0 0 - expect division

			by 0" **/: division by zero**	by 0 **0.0**
AssignmentOneNumberSix	30	72	"Test case: Testing 1 2 3 – expect #t" **#t** "Test case: Testing 1 1 2 – expect #f" **#f** "Test case: Testing 1 1 1 – expect #f" **#f**	Test case: Testing 1 2 3 - expect true **true** Test case: Testing 1 1 2 - expect false **false** Test case: Testing 1 1 1 - expect false **false**
AssignmentOneNumberSix Variation	16	71	"Testing allSame 10 10 10, expecting true" **#t** "Testing someDifferent 10 10 10, expecting false" **#f**	Testing allSame 10 10 10, expecting true **true** Testing someDifferent 10 10 10, expecting false **false**
AssignmentTwoSnippets	42	153	"Testing length of 0, expecting 1" **1**	Testing length of 0, expecting 1 **1**
neg	12	35	"Testing neg 10, expect -10" **-10** "Testing neg -10, expect 10" **10**	Testing neg 10, expect -10 **-10.0** Testing neg -10, expect 10 **10.0**
genericFunction	9	54	" What is f using 10 20? Expecting 2" **2**	What is f using 10 20? Expecting 2 **2.0**
largestWithIfStatement	25	56	"What is the largest of 10 and 20? Expecting 20" **20** "What is the largest of -10 and -20? Expecting -10" **-10**	What is the largest of 10 and 20? Expecting 20 **20** What is the largest of -10 and –20? Expecting -10 **-10**

			"What is the largest of 10 and 2+20? Expecting 22" **22**	What is the largest of 10 and 2+20? Expecting 22 **22.0**
isEvenWithIfStatementAnd Functions	53	104	"Testing if 132 is even" **#t** "Testing if 0 is even" **#t** "Testing if 11 is even" **#f** "Testing if -99999 is even" **#f** "Testing halve-or-zero 2, expecting back 1" **1** "Testing halve-or-zero 113, expecting back 0" **0**	Testing if 132 is even **true** Testing if 0 is even **true** Testing if 11 is even **false** Testing if -99999 is even **false** Testing halve-or-zero 2, expecting back 1 **1.0** Testing halve-or-zero 113, expecting back 0 **0**
usingSetBang	15	42	"setting x to a constant, expecting x to be 5" **5** "resetting setting x to a function, expecting x to be 65" **65**	setting x to a constant, expecting x to be 5 **5** resetting setting x to a function, expecting x to be 65 **65.0**
transformingConds	24	65	"Which is larger, 10 or 5?" **10** "Which is larger, 10 or 15?" **15**	Which is larger, 10 or 5? **10** Which is larger, 10 or 15? **15**

			"Which is larger, 10 or 10?" "**equal**"	Which is larger, 10 or 10? **equal**
deepRecursion	9	63	"testing with 10" **-98**	testing with 10 **-98.0**
lambdaDouble	59 [*irregular formatting*]	46 [*regular formatting*]	"What is the result of 5 passed into the lambda?" **10** "What is the result of -10 passed into the lambda?" **-20**	What is the result of 5 passed into the lambda? **10.0** What is the result of -10 passed into the lambda? **-20.0**
defineVariableBinding	11 [*irregular formatting*]	30 [*regular formatting*]	"Calling function with value 3, expecting back 6" **6**	Calling function with value 3, expecting back 6 **6.0**
lambdaWithVariables	38	61	"Global variable is: "**3** "Calling Lambda with 1 and 2, expecting back 1 + 2 + 3 = 6..." "Lambda is called" "Adding "**1**" and "**2**" and the global variable, "**3** "Result is "**6** "Returning: "**6**	Global variable is: **3** Calling Lambda with 1 and 2, expecting back 1 + 2 + 3 = 6... Lambda is called Adding **1** and **2** and the global variable, **3.0** Result is **6.0** Returning: **6.0**
lambdaWithSetVariablesError	45	72	"Global variable is: "**3** "Calling Lambda with 1 and 2, expecting back 1 + 2 + 3 = 6..." "Lambda is called" "Adding "**1**" and "**2**" and the global variable, "**3**	Global variable is: **3** Calling Lambda with 1 and 2, expecting back 1 + 2 + 3 = 6... Lambda is called Adding **1** and **2** and the global variable, **3.0**

			"Result is "**6** "Returning: "**6** "Setting globalvar to **10**" "Now calling, expecting back 1 + 2 + 10 = 13" "Lambda is called" "Adding "**1**" and "**2**" and the global variable, "**10** "Result is "**13** "Returning: "**13**	Result is **6.0** Returning: **6.0** Setting globalvar to **10** Now calling, expecting back 1 + 2 + 10 = 13 Lambda is called Adding **1** and **2** and the global variable, **3.0** Result is **6.0** Returning: **6.0**
usingQuote	6	15	**this_is_with_a_ quote this_is_with_ using_the_short hand (+ 2 3)**	**This_is_with_a _quote This_is_with_ using_the_ shorthand (+ 2 3)**

Table 5.1.6 – Raw Output and Line Counts of All Testing

5.2 Appraisal of Results

The results are good. The original goal of functionally-equivalent Scheme to Java source translation has been achieved at the level I wanted. As a benchmark for the complexity of the Scheme code being translated, I have been using the Scheme assignments from 95.307, the course in which functional programming is taught at Carleton University. Incidentally, this is also the reason for the choice and version of the Scheme interpreter employed. My rationale in this was that these assignments should provide a good overview of the commonly-used constructs, and suggest a level of complexity appropriate for my project. I have judged the value and completeness of this project in respect to the assignments of the course. My goal was to translate the majority of assignment 1 – I feel this goal was met. Translating assignment 2 would be nice, and some steps were made in that direction. The holy grail of this project – a goal I did not expect to reach – would have been translation to Java of the

meta-circular interpreter implemented in assignment 3. Scheme2Java can translate most of assignment 1 (see the disk for full conversions) and some of assignment 2, such as the helper functions in the Scheme objects. However, the object-oriented Scheme of assignment 2 does not translate fully, nor does the function-passing in assignment 1. I consider Scheme2Java to be beta or experimental software. It is not ready for public release.

I am pleased with these results, but they are not as spectacular as I'd hoped. This is because of an unforeseen consequence of using the Scheme-atomic model: individual *Translation* objects have very little idea of the state and layout of the Scheme program around them.

What this means is that while "(+ 2 3)" does correctly translate, it does so in the most basic and verbose way possible. A human being translating that command would change it to something like "int result = 2 + 3;" in Java, a far cry from the 3-line Scheme2Java translation in **Figure 4.3.1**. Over the course of longer, more complex programs, the translation becomes more and more verbose. There is an average increase of two and a half lines of code per line of Scheme code (this of course depends on how the Scheme code was formatted and what it is doing), which leads to the bloat of the resulting Java code. A lot of this comes from functions. In the Java code, translation for a simple function requires at least three lines of overhead: the function definition and open bracket, the return statement, and the closing bracket. The Scheme code this is translated from has only two characters of overhead: the open and close brackets.

Some level of post-translation which could take these Java structures and compress them would be recommended future work. In fact, at this level, it might be better to return meta-information on the translated Scheme code, rather than Java code itself. This would allow easier organization of the code before the final output.

There are a few restrictions on the Scheme code as a result of this atomic model. The most critical one – which could be resolved with having the atoms know more of their context – is that Scheme functions must be defined in the order they are used. If a function called "halve" calls a function called "is-even", "is-even" must be in the code before "halve", otherwise the translation fails. An alternate way to work around this problem would be to identify all the function calls in a pre-translate method and keep those in mind while translating.

The translation of lambdas is decent, but not splendid. The problem with the translation approach for lambdas is that once they are encountered in the Scheme code, they are translated to unchanging Java code, environment and all. This allows the lambda to accurately represent the environment of the Scheme lambda at calling time, but should that state change, the lambda may return different results. For instance, in the example "LambdaWithSetVariablesError" on the CD, the lambda uses its own internal environment (the one "frozen" when it was created) once a variable in its environment has its value changed by the set! command. This was desired behavior, as it was how my Scheme execution model behaved. This however does differ from Dr. Scheme's execution model, and this difference only became apparent late in the development process. And unfortunately, the methods to make the two agree are not trivial. I could re-write the Translation model so that in the translated Java source, variables are stored not as Scheme SVars but rather in some data structure, that lambdas (and indeed all other Scheme object types) access when necessary. This would involve a pretty fundamental

rewrite. However, it's better than the alternative, wherein I pre-translate so that all calls to a pre-defined lambda have the code for the lambda copied in, in place of the call. This would allow a separate lambda to be created for each call, solving this problem but introducing further ones, as well as greatly adding to the Java code bloat. Neither option has been implemented, so lambda translation must be considered incomplete or experimental. However, it does work in the majority of cases where the outside environment doesn't change between executions.

Chapter 6

Conclusions

A summary of conclusions drawn from this project is presented and some suggestions towards future work are made in this chapter.

6.1 Results Achieved

Scheme to Java source translation is possible and practical: this program represents the first step in a source-to-source conversion. While it is far from R^5RS completeness, it does allow conversion of many basic and useful Scheme constructs, and can easily serve as a base for future work. The program is a success, and outputs functionally-equivalent Java source code. However, more concise code, something more akin to what a human developer would write, is desired.

Relative to the other options in the field, I believe my program offers something of value. Of course it is not as complete as Kawa, but it does offer valuable and clear source-to-source conversion: something Kawa doesn't do. Furthermore, while this is experimental software, it does have practical worth and can be used for real Scheme to Java conversions. Its code is also designed for clarity and ease-in-understanding, an aid to future development by other programmers.

This is the first source-to-source Scheme to Java translator ever, and I believe it is a good one.

6.2 Future Work

6.2.1 A More Complete Command Set

Obviously, adding more commands to the list of supported commands in Scheme2Java is some future work than can be done. I have been greedy and done most of the interesting (ie: commonly used and interesting-to-translate) commands, but there are still some good ones to do: *case* (this would be a pre-transform into a Scheme *if*, like *cond*) *cons*, *cdr*, *car* (these would likely be transformed into special Java objects), *let*, etc. The more supported commands from the R[5]RS set, the better.

6.2.1 A More Intelligent Post-Translation

As alluded to before, a post-translation method wherein the outputted Java code was compressed would be useful. Such a method might even imply a redesign on the translation process, so that some other representation of the translated Java code would be returned instead of a string. Originally, I had intended to return *TranslatedScheme* objects, which would contain enough information to generate the Java code, but would not be the actual Java code. This would allow code to be rearranged through easier methods than String manipulation. However, I realized that a linked-list of Strings could provide the same functionality, and at the level of translation I'm doing, such rearranging isn't typically necessary.

Another option I considered was to return XML representations of the Java code, containing both the Java code and meta information about the Java code. This however was clearly a bad idea, as XML can be mapped one-to-one to an object: this would be XML for XML's sake. Besides, the above objections to an object still applied. There are proprietary applications that generate Java code from XML, but this would be an unnecessary layer, as I can already generate the Java code myself.

The returns on programming a simplifying post-translate method might not be as big as one might expect, as there are already utilities (many available for free) that optimize compiled Java code[5]. I didn't test it, but I believe these programs might provide comparable optimization to what can be accomplished through a method in this program. Of course there is no automated antidote or fix to bad code, but the output of Scheme2Java isn't bad, just repetitive. It should succumb to some basic optimizations.

6.2.3 More Context for Translation Objects

Another option for future work would be to change the atomic model so that atoms have a better idea of their place in the larger program. This would result in better code and be an interesting programming task, as representing the context of a function in Scheme from Java's perspective would be non-trivial. It would also allow the variable storage used for lambdas to be more robust than it is now. Presently, identically-named variables at identically-deep levels in the Scheme code may not be recorded properly in a lambda accessing them.

6.2.4 Translation to C++ and Other Languages

Interesting future work would be to adapt the software to translate to C++ as well as Java. This would not be too difficult, and it would result in nicer, operator-overloaded C++ output as opposed to the more limited output of the Java language. *Translator* objects could either return code for both (allowing the user to choose), or, more interestingly, meta-information on the code to be generate, which could then be outputted via a output(lang) method, passing in either "C++" or "Java", or any other supported language.

[5] One such list available here: http://www.geocities.com/marcoschmidt.geo/java-class-file-optimizers.html

6.2.5 Other Future Work

Basic advances for further work also include a GUI – this is uninteresting to develop but could be handy. The CLI provides all functionality that a GUI would. Also, one feature that I would have liked to program would be a check to make sure that the Scheme input code contains no reserved Java words. For instance, a function called *double* in Scheme is illegal in Java, such names should be replaced by the translator before the translation begins. Also, because of the way the translation proceeds, functions with the same name as variables can be translated incorrectly. A simple transformation to make all names unique could be implemented.

Additionally, lambda handling is not as great as it could be. On their own they are handled fine, however, passing lambdas into functions isn't identified properly. Code like this fails:

```
(define (a-plus-abs-b a b)
   ((if (> b 0) + - ) a b))
```

This is because + and – aren't identified as lambdas and handled as such. This is a more complex problem than it initially appeared to be, and I'm sorry I didn't have more time to explore it more fully. A student continuing work on this project would have their first goal defined as developing more robust handling of lambdas. The present work-around is to not write Scheme code like that, but rather equivalent Scheme code:

```
(define (a-plus-abs-b a b)
  ((if (> b 0)
       (+ a b)
       (- a b)
   )
 )
```

References

Bothner, Per. "Kawa, the Java-based Scheme system". Posted 12 June 2002. Accessed 20 January 2003. <http://www.gnu.org/software/kawa/>

Raj, Gopalan Suresh. "The Factory Method (Creational) Design Pattern". Posted 27 Dec 2002. Accessed 27 Feb 2003. <http://gsraj.tripod.com/design/creational/factory/factory.html>

Richard Kelsey, William Clinger, and Jonathan Rees. "Revised^5 Report on the Algorithmic Language Scheme". Posted 20 February 1998. Accessed 2 January 2003. <http://www.schemers.org/Documents/Standards/R5RS/HTML/>

Schmidt, Marco. "Java class file optimization and compression tools". Posted 19 December 2002. Accessed 14 February 2003. <http://www.geocities.com/marcoschmidt.geo/java-class-file-optimizers.html>

Sun Microsystems. "Java™ 2 Platform, Standard Edition, v 1.4.1 - API Specification". Posted 2002. Accessed 1 March 2003. <http://java.sun.com/j2se/1.4.1/docs/api/index.html>

Appendix A

Competing Options for Scheme Translation

There are several Scheme in Java projects available on the Internet, with various implementation strategies. All are either implementations of the Scheme language or direct-bytecode converters; I could find no translators. Alphabetically:

Jaja: Scheme in Java

http://youpou.lip6.fr/queinnec/Java/Jaja.html

Closed source, and no longer active: a Scheme interpreter in Java, done partially as an example of OO coding.

Jscheme by Peter Norvig (previously "Silk")

http://www.norvig.com/jscheme.html

An open-source implementation of Scheme in Java, with some extensions. This takes Scheme source code and converts it to its own Java objects, which it then executes.

Kawa, the Java-based Scheme system

http://www.gnu.org/software/kawa/

The most advanced of the converters, and the only one that isn't an interpreter. Kawa does not convert Scheme into Java source; rather, it converts it onto its own internal Java "Expression" objects, which are then compiled to Java bytecode and executed.

Scheme For Java

http://student.nada.kth.se/~d93-hyo/scheme/

Another open-source but inactive Scheme interpreter for Java.

Skij

http://alphaworks.ibm.com/tech/Skij

Apparently abandoned (the last update was in March 1999), Skij is a corporate-developed Scheme interpreter, written and running in Java.

There are also several projects for which only broken links can be found as evidence of their existence.

Appendix B

Annotated Conversions

Rather than clutter the rationale and logic behind the conversions with typically-verbose examples, included here are actual inputs to and outputs of the Scheme2Java program, along with explanations.

On the following pages you will find these examples of conversion:

Conversion 1 - Shows a simple math conversion and define binding.
Conversion 2 - Shows recursion down between functions, and how the required variable is passed along as deep as is needed.
Conversion 3 - Shows how the Java code can become bloated, compared to the Scheme code.
Conversion 4 - Shows the transformation of an if statement.
Conversion 5 - Shows how lambda is translated.
Conversion 6 - Shows how lambda is translated when the environment needs things stored.

Included on the CD as **Appendix D** are the complete source and .class files for all these examples, among others. In the source for all the Scheme examples, you will find brief comments explaining the purpose of the example and anything noteworthy in the output. It is recommended that anyone interested in the operation of Scheme2Java explore these examples.

B.1 Conversion 1

Rationale:
Shows a simple math conversion and define binding.

Input:
```
(define (neg x)
        (- x x x)
)
```

Output:
```
//Generated Thu Apr 10 13:31:36 EDT 2003;
import java.io.*;
import java.util.*;
public class B1 {
        private SVar mathvar4(SVar x){
                SVar var6 = new SVar(x);
                SVar var8 = new SVar(x);
                SVar var10 = new SVar(x);
                SVar var3 = new SVar(var6.toDouble() - var8.toDouble() –
                        var10.toDouble() );
                return var3;
        }
        public SVar neg(SVar x){
                return mathvar4(x);
        }
        public void run(){
        }
        public static void main(String[] args){
                B1 ts = new B1();
                ts.run();
        }
}
//EOF - B1
```

Remarks:
Notice that this program, when run, doesn't do anything. That's because there is no code to execute in the Scheme input. Had the line (neg 2) been added, the output Java code would instead have the run method:

```
        public void run(){
                SVar var1 = new SVar("2");
                neg(var1);
        }
```

This can be seen in the other examples.

B.2 Conversion 2

Rationale:
Shows recursion down between functions, and how the required variable is passed along as deep as is needed.

Input:
```
(define (awesome x)
        (+ 2 (- 3 (* 4 (/ 5 (* 100 241241241246 x)))))
)
(write (awesome 10))
```

Output:
```
//Generated Thu Apr 10 13:33:29 EDT 2003;
import java.io.*;
import java.util.*;
public class B2 {
        private SVar mathvar20(SVar x){
                SVar var22 = new SVar("100");
                SVar var24 = new SVar("241241241246");
                SVar var26 = new SVar(x);
                SVar var20 = new SVar(var22.toDouble() * var24.toDouble() *
                        var26.toDouble() );
                return var20;
        }
        private SVar mathvar16(SVar x){
                SVar var18 = new SVar("5");
                SVar var16 = new SVar(var18.toDouble() / mathvar20(x).toDouble()
                        );
                return var16;
        }
        private SVar mathvar12(SVar x){
                SVar var14 = new SVar("4");
                SVar var12 = new SVar(var14.toDouble() * mathvar16(x).toDouble()
                        );
                return var12;
        }
        private SVar mathvar8(SVar x){
                SVar var10 = new SVar("3");
                SVar var8 = new SVar(var10.toDouble() - mathvar12(x).toDouble() );
                return var8;
        }
        private SVar mathvar4(SVar x){
                SVar var6 = new SVar("2");
                SVar var3 = new SVar(var6.toDouble() + mathvar8(x).toDouble() );
```

```
            return var3;
        }
        public SVar awesome(SVar x){
            return mathvar4(x);
        }
        public void run(){
            SVar var32 = new SVar("10");
            SVar var30 = awesome(var32);
            System.out.print(var30);
        }
        public static void main(String[] args){
            B2 ts = new B2();
            ts.run();
        }
}
//EOF - B2
```

Remarks:
The "x" variable is passed into the function "awesome', even though it is not used in that function, but rather in a function called by a function used there. Unlike the previous example, when executed this Java code actually does something: print out the output of the function.

Also note how there is one function per nesting-level of the Scheme code, and how the math conversion can handle any number of inputs: (+ n_1, n_2, n_3, …, n_i).

Additionally, the translator recognizes that "awesome" is a reference to a previously-translated function.

B.3 Conversion 3

Rationale:
Shows how the Java code can become bloated, compared to the Scheme code.

Input:
```
(define (f a b)
        (/(+ (modulo a 2) 5)  (* 9 (abs b)))
)
(write (f 10 20))
```

Output:
```
//Generated Thu Apr 10 13:35:06 EDT 2003;
import java.io.*;
import java.util.*;
public class B3 {
        private SVar mathvar8(SVar a){
                SVar var10 = new SVar(a);
                SVar var12 = new SVar("2");
                SVar var8 = new SVar(var10.toDouble() % var12.toDouble() );
                return var8;
        }
        private SVar mathvar6(SVar a){

                SVar var14 = new SVar("5");
                SVar var6 = new SVar(mathvar8(a).toDouble() + var14.toDouble() );
                return var6;
        }
        private SVar mathvar20(SVar b){
                SVar var22 = new SVar(b);
                SVar var20 = new SVar(Math.abs(var22.toDouble()) );
                return var20;
        }
        private SVar mathvar16(SVar b){
                SVar var18 = new SVar("9");
                SVar var16 = new SVar(var18.toDouble() * mathvar20(b).toDouble()
                        );
                return var16;
        }
        private SVar mathvar4(SVar a, SVar b){

                SVar var3 = new SVar(mathvar6(a).toDouble() /
                        mathvar16(b).toDouble() );
                return var3;
        }
```

```
	public SVar f(SVar a , SVar b){
		return mathvar4(a, b);
	}
	public void run(){
		SVar var28 = new SVar("10");
		SVar var30 = new SVar("20");
		SVar var26 = f(var28, var30);
		System.out.print(var26);
	}
	public static void main(String[] args){
		B3 ts = new B3();
		ts.run();
	}
}
//EOF - B3
```

Remarks:
This is the bloat of Java that I was talking about. 3 lines of Scheme code become 46 lines of Java code: over 15 times the size. Notice how the recursion works: the nested functions in Scheme become separate functions in Java, called when required. Also, notice how here as well, “a” and “b” are passed along until needed, and “f” is recognized as a call to a previously-translated function.

B.4 Conversion 4

Rationale:
Shows the transformation of an if statement.

Input:
```
(define (largest x y)
        ;returns largest of 2 numbers
        (if ( >= x y )
                x
                y
        )
)
(write (largest 10 20))
```

Output:
```
//Generated Thu Apr 10 13:36:35 EDT 2003;
import java.io.*;
import java.util.*;
public class B4 {
        private SVar mathvar6(SVar x, SVar y){
                SVar var8 = new SVar(x);
                SVar var10 = new SVar(y);
                SVar var6 = new SVar(var8.toDouble() >= var10.toDouble() );
                return var6;
        }
        public SVar largest(SVar x , SVar y){
                SVar var14 = new SVar(y);
                SVar var12 = new SVar(x);
                SVar var3 = null;
                if (mathvar6(x, y).toBoolean()){
                        var3 = var12;
                }else{
                        var3 = var14;
                }
                return var3;
        }
        public void run(){
                SVar var20 = new SVar("10");
                SVar var22 = new SVar("20");
                SVar var18 = largest(var20, var22);
                System.out.print(var18);
        }
        public static void main(String[] args){
                B4 ts = new B4();
```

```
                ts.run();
        }
}
//EOF - B4
```

Remarks:
Notice how the if statement is actually the simplest code: all the difficult work is done by the other translators, recursively.

B.5 Conversion 5

Rationale:
Shows how lambda is translated.

Input:
(define tdouble (lambda (a) (+ a a)))
(tdouble 4)

Output:
```
//Generated Thu Apr 10 13:38:31 EDT 2003;
import java.io.*;
import java.util.*;
public class B5 {
	private SVar mathvar6(SVar a){
		SVar var8 = new SVar(a);
		SVar var10 = new SVar(a);
		SVar var5 = new SVar(var8.toDouble() + var10.toDouble() );
		return var5;
	}
	private class lambdavar5{
		private SVar execute(SVar a){
			return mathvar6(a);
		}
	}
	public void run(){
		lambdavar5 tdouble =  new lambdavar5();
		SVar var14 = new SVar("4");
		SVar var12 = tdouble.execute(var14);
	}
	public static void main(String[] args){
		B5 ts = new B5();
		ts.run();
	}
}
//EOF - B5
```

Remarks:
lambdavar5 is a separate class, with its environment stored. Here there is nothing necessary to store, so all that is required is the execute() method. When the lambda translator returns, it returns information indicating both that it does represent a lamda, and that this lambda requires so many variables in order to execute.

B.6 Conversion 6

Rationale:
Shows how lambda is translated when the environment is stored. Also highlights the method in which the environment is stored.

Input:
```
(define global 0)
(set! global 10)
(define add_to_global (lambda (a) (+ a global)))
(write (add_to_global 4))
```

Output:
```
//Generated Thu Apr 10 13:44:47 EDT 2003;
import java.io.*;
import java.util.*;
public class B6 {
	private SVar mathvar14(SVar a, SVar global){
		SVar var16 = new SVar(a);
		SVar var18 = new SVar(global);
		SVar var13 = new SVar(var16.toDouble() + var18.toDouble() );
		return var13;
	}
	private class lambdavar13{
		private SVar execute(SVar a){
			return mathvar14(a, global);
		}
		SVar var7 = new SVar(10);
		SVar var3 = new SVar(0);
		SVar global = new SVar(var7);
	}
	public void run(){
		SVar var3 = new SVar("0");
		SVar global = var3;
		SVar var7 = new SVar("10");
		global  = new SVar(var7);
		lambdavar13 add_to_global =  new lambdavar13();
		SVar var24 = new SVar("4");
		SVar var22 = add_to_global.execute(var24);
		System.out.print(var22);
	}
	public static void main(String[] args){
		B6 ts = new B6();
		ts.run();
	}
```

```
}
//EOF - B6
```

Remarks:
Notice how lambdavar13 now has both an execute method and a list of variables in the environment. Not all are used. In the example, we first defined global to be 0, then used set! to redefine it to 10. The variables in lambdavar13 reflect this, having both a variable for 0 and 10. Notice however that only the most recent value, 10, is bound to the variable "global".

All variables when created get added to the environment, and the side-effect of the atomic model is that many temporary variables (var7, var3) get added that may not be used when the lambda is executed.

Appendix C

Instructions for Running the Software

Scheme2Java is run from the command line, via the *Scheme2JavaCLI*:

To run, switch to the directory containing the Scheme2Java binaries. Run Scheme2Java as follows:

java Scheme2JavaCLI [input file] [classname] [logging level] [javac path]

where

[input file] is the Scheme code you would like translated
[classname] is the name of the output class you'd like. This can be any one-word string, and should be Java-valid, otherwise the source won't compile. The output will be stored in a file called **[classname]**.java
If the file exists, it will be overwritten without warning.
[logging level] is the amount of logging you'd like printed to the screen. This is a number from 3 to 1.

At level 3 everything is printed out, including debug information.
At level 2 everything important is printed out, enough so that you can follow the translation process by hand.
At level 1 only the identification of the commands, and the final output, is printed.

[javac path] is the path to the Javac compiler. Providing a path here indicates that you'd like the output source to be compiled after translated. This parameter is optional.

Compiled output can be run by invoking "java **[classname]**".

Also included is a helper batch file, called go.bat. go.bat allows you to specify only the classname you want – it then translates, compiles, and runs. This lets you see output immediately.

To use go.bat, first edit it so that it points to your installation of javac.exe. The default location for this file is c:\j2sdk1.4.1_01\bin\, and that's where it looks unless edited otherwise.

Then, invoke go.bat as follows:

go **[classname]**

Your Scheme code will be translated, compiled, and will be run.

Note that to compile and run the Java-translationed code, you need to have SVar.class in the same directory or in java's classpath.