Embedding Scheme in Java

by

Brian D. Carlstrom

S.B., Massachusetts Institute of Technology (1995)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2001

Author . . . .
Department of Electrical Engineering and Computer Science
February 6, 2001

Certified by . . . .
Dr. Olin Shivers
Research Scientist, Artificial Intelligence Laboratory
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Embedding Scheme in Java

by

Brian D. Carlstrom

Submitted to the Department of Electrical Engineering and Computer Science
on February 6, 2001, in partial fulfillment of the
requirements for the degree of
Master of Engineering

Abstract

Extension languages are an important part of modern applications development. Java as a platform does not provide a standard extension language. Scheme is one possible choice as an extension language for Java. There are a variety of techniques for implementing Scheme in Java varying from interpreting s-expressions to compiling into Java byte-codes. The historical evolution of one implementation is discussed over the course of several years. The design of the Java-to-Scheme and Scheme-to-Java interfaces is reviewed. The advantages and disadvantages of Java and Scheme are compared.

Thesis Supervisor: Dr. Olin Shivers
Title: Research Scientist, Artificial Intelligence Laboratory

Contents

1 Introduction **15**

2 Interpretation Strategies **17**

2.1 Expression Interpreter 17

2.2 Statement Interpreter 18

2.3 Byte-code Interpreter 18

2.4 Byte-code Generation 18

3 First-Pass Implementation **21**

3.1 Beginning 21

3.2 `Expression.eval` 22

3.3 `Procedure.apply` 23

3.4 Syntax 24

3.5 Scheme types and their Java representation 25

3.5.1 `SelfEvaluating` 26

3.5.2 `booleans` 26

3.5.3 `symbols` 26

3.5.4 `numbers` 26

3.5.5 `characters` 26

3.5.6 `strings` 27

3.5.7 `pairs` 27

3.5.8 `vectors` 27

3.5.9 `procedures` 28

3.5.10 `ports` 28
3.6 `Reader` 28
3.7 `Expression.analyze` 28
3.8 `Loader` 29
3.9 `Writer` 30
3.10 Primitives 30
3.11 `Script` 31
3.12 `REPL` 31
3.13 `ScriptException` 32
3.14 Java-to-Scheme API 32
3.15 Extensions to Scheme for Java 33
3.15.1 `java.lang.Object` 33
3.15.2 `java.util.*` 35
3.15.3 Processes 36
3.15.4 Mail 36
3.16 Analysis of First-Pass Implementation 36
3.16.1 Performance 36
3.16.2 Maintainability 38
3.16.3 Standard Compliance 38

4 Second-Pass Implementation **41**
4.1 Removing `SelfEvaluating` `Expression` 41
4.1.1 `Expression` Inheritance Cleanup 42
4.1.2 Primitive Type Marshalling 44
4.2 `Compiler` 45
4.2.1 `CompileTimeEnvironment` 46
4.2.2 `GlobalVariables` as Cells 46
4.2.3 Table-Driven Syntax 48
4.3 Primitives 50
4.3.1 I/O Primitives 51

4.3.2 Externalizing Primitive Definitions 52
4.3.3 Removing Non-Primitive Primitives 53
4.3.4 Partitioning Primitive Definitions 54
4.4 Arrays 54
4.4.1 `StringBuffer` to `char[]` 55
4.4.2 Arguments from `Vector` to `Object[]` 55
4.5 `Application` Special Cases 55
4.5.1 Unrolling Primitives 58
4.6 Handling of `Exceptions` 59
4.7 Debugging 59
4.7.1 Java Debugger 60
4.7.2 Stack Traces 61
4.7.3 Source 62
4.7.4 `REPLServer` 63
4.8 Analysis of Second-Pass Implementation 63
4.8.1 Modules 64
4.8.2 Performance 64
4.8.3 Macros 66

5 Third-Pass Implementation 67

5.1 `let` Optimization 67
5.2 Closure Analysis 69
5.2.1 `Stack` 70
5.2.2 `Until` 71
5.2.3 Closure Analysis at Compile Time 72
5.2.4 Closure Analysis at Run Time 74
5.3 `Quoted` 75
5.4 Removing Implicit `Begin` 75
5.5 Analysis of Third-Pass Implementation 76
5.5.1 Analysis of `let` Optimization 76

5.5.2 Analysis of Closure Analysis 77

6 Fourth-Pass Implementation **79**

6.1 `Applet` 79

6.1.1 `java.net.URL` 79

6.1.2 Syntax Checking 80

6.1.3 `ScriptException` 80

6.1.4 Script Widget 81

6.2 Reflection 81

6.2.1 `java.lang.reflect` 81

6.2.2 Reflection Extensions 82

6.2.3 Reflection Performance 84

6.3 Multi-engine 85

6.3.1 `Procedures` 86

6.3.2 Thread-Local Storage versus `Stack` 87

6.4 Internationalization 87

6.5 Performance 88

6.5.1 `GrowOnlyHashtable` 88

6.5.2 `new Integer` 88

6.5.3 `char[]` to `String` 89

6.6 Analysis of Fourth-Pass Implementation 89

6.6.1 `Applet` versus Reflection 89

6.6.2 Primitives in `Applet` Environment 90

6.6.3 Multi-Engine versus `REPLServer` versus HTML 91

6.6.4 Remaining Limitations to Scheme for Java 92

7 Java and Scheme **95**

7.1 Java Advantages 95

7.1.1 Portability 95

7.1.2 Language 97

7.1.3 Platform 101

7.2 Java Disadvantages 102
7.2.1 Threads 102
7.2.2 Synchronization 103
7.2.3 Classes 106
7.2.4 `RuntimeExceptions` 108
7.2.5 Assert and Macros 109
7.2.6 `Numbers` 109
7.2.7 `else if` 109
7.2.8 Exit 110
7.2.9 Tail Recursion 110
7.3 Scheme Advantages 111
7.3.1 Size 111
7.3.2 Garbage Collection 111
7.3.3 Functional Programming 111
7.4 Scheme Disadvantages 112
7.4.1 Language 112
7.4.2 Libraries 116
7.4.3 I/O 118
7.4.4 Platform 120
7.4.5 Testing 120
7.4.6 Goals 121

8 Language Discussion **123**
8.1 Code-Data Duality 123
8.2 Packages and Modules 124
8.3 Type Safety 125
8.4 Dynamic Invocation 126
8.4.1 C 126
8.4.2 Scheme 127
8.4.3 Java 127

8.5 Threads, Dynamic Variables, and Thread-Local Storage 128
8.6 Syntax 129

9 Comparative Analysis 131

9.1 Comparative Analysis with other Scheme systems 131
9.1.1 Java Scheme Systems 132
9.1.2 Non-Java Scheme Systems 138
9.2 Comparative Analysis with other Scheme-like Java systems 143
9.2.1 The scheme package 143
9.2.2 PS3I 143
9.2.3 LISC 143
9.2.4 HotScheme 144
9.2.5 MIT Scheme in Java 144
9.2.6 PAT 144
9.2.7 LispkitLISP Compiler in Java 144
9.3 Comparative Analysis with other Java extension systems 145
9.3.1 HotTea 145
9.3.2 Rhino 145
9.3.3 Jacl 145
9.3.4 JPython 146
9.3.5 BeanShell 146
9.3.6 DynamicJava 146

10 Future Work 147

11 Conclusion 149

11.1 Scheme-to-Java API 149
11.2 Java-to-Scheme API 150
11.3 Java Performance Lessons 152
11.3.1 Thou shall not synchronize 152
11.3.2 Thou shall not allocate 153

11.3.3 Thou shall not abuse exceptions . . . 153
11.3.4 Thou shall not forsake buffering . . . 154
11.3.5 Thou shall not forsake arrays . . . 154
11.3.6 Thou shall honor pointer equality . . . 154
11.4 Final Thoughts . . . 155

12 Acknowledgments **157**

A Benchmark Results **159**
A.1 Script . . . 160
A.1.1 Sun JDK 1.3.0 . . . 160
A.1.2 Sun JDK 1.2.2 . . . 160
A.1.3 Sun JDK 1.1.8 . . . 160
A.1.4 IBM JDK 1.3.0 . . . 161
A.1.5 IBM JDK 1.1.8 . . . 161
A.1.6 Microsoft SDK for Java 3.1 . . . 161
A.2 Kawa . . . 162
A.2.1 Sun JDK 1.3.0 . . . 162
A.2.2 Sun JDK 1.2.2 . . . 162
A.2.3 Sun JDK 1.1.8 . . . 162
A.2.4 IBM JDK 1.3.0 . . . 163
A.2.5 IBM JDK 1.1.8 . . . 163
A.2.6 Microsoft SDK for Java 3.1 . . . 163
A.3 SILK . . . 164
A.3.1 Sun JDK 1.3.0 . . . 164
A.3.2 Sun JDK 1.2.2 . . . 164
A.3.3 Sun JDK 1.1.8 . . . 164
A.3.4 IBM JDK 1.3.0 . . . 165
A.3.5 IBM JDK 1.1.8 . . . 165
A.3.6 Microsoft SDK for Java 3.1 . . . 165
A.4 Skij . . . 166

A.4.1 Sun JDK 1.3.0 166
A.4.2 Sun JDK 1.2.2 166
A.4.3 Sun JDK 1.1.8 166
A.4.4 IBM JDK 1.3.0 167
A.4.5 IBM JDK 1.1.8 167
A.4.6 Microsoft SDK for Java 3.1 167
A.5 WinScheme based on Scheme 48 0.52 168
A.5.1 without ,bench 168
A.5.2 with ,bench 168
A.6 SCM 5d3 169
A.7 MIT Scheme 7.5.10 170
A.7.1 plain load 170
A.7.2 sf load 170
A.7.3 cf load 170

List of Tables

3.1 `Expression` subclasses 22

3.2 Scheme types and their Java representation 25

3.3 ScriptException subclasses 32

4.1 Variable and Assignment replacement Expression subclasses 46

Chapter 1

Introduction

Extension languages are an important part of modern applications development. They allow the end user to tailor an application to needs that could not be foreseen by the developer. Early examples of extension languages were often tied directly to one application, as is the case with Emacs Lisp in GNU Emacs. [32] A later trend was to provide an extension language as a reusable library, as is the case with Tcl/Tk. [34] [35] Recently the trend has been to provide an interface between applications that desire scripting and libraries that can provide it, allowing users to use their language of choice, as is the case with ActiveX Scripting. [42]

Java provides a new twist for extension languages. A pure Java application cannot use any of the non-Java extension languages without compromising portability. However, a new extension language built in Java would inherit some of its parent language's benefits, such as cross platform support, modern garbage collector technology, and just-in-time compiler support. [16]

Scheme is a good choice as an extension language for Java. Scheme is a small well-defined language making it easier on language users and language implementors alike. Although Scheme is small, it is a general-purpose programming language providing traditional data-structures as well as object-oriented techniques. Scheme's data-structures are easily represented by standard Java classes making interoperability straightforward. [28]

After discussing possible implementation strategies, the history of one particular

Scheme in Java system will be discussed. This system had four discrete implementation passes, each with a different motivations:

1. minimal quick implementation and simple embedding API
2. maturation of libraries and simple performance optimizations
3. serious performance work based on application memory and CPU profiling
4. full-featured embedding API and focus on Java environment support

Each pass will offer analysis of the implementation at that point in time. A language implementation faces various tradeoffs between run-time speed, run-time memory usage, implementation size, complexity, extensiblity, usability, and even correctness, these will be reviewed in their historical context.

This will be followed up by a pros and cons discussion of the Java and Scheme programming languages as well as more general thoughts on programming languages. Finally, comparative analysis, future work, and conclusions are presented.

Chapter 2

Interpretation Strategies

A variety of implementation techniques exists implementing Scheme in Java, varying from interpreting s-expressions to compiling into Java byte-codes. Tradeoffs exist for each approach, such as speed, size, and implementation complexity. Ruling out the extremes of a simple s-expression interpreter for its unnecessarily poor analysis and a Java byte-code system for its complexity, a suitable strategy must lie somewhere in between.

2.1 Expression Interpreter

An expression interpreter is one step up from an s-expression interpreter. This uses a simple compiler to do syntax analysis as well as translation of derived syntax into a smaller kernel syntax. Expressions in this kernel syntax would be represented directly by subclasses of a Java class `Expression` that would implement an `eval` method. Procedures would in turn be represented by subclasses of a Java class `Procedure` that would implement an `apply` method. Such an interpreter could also do traditional lexical analysis to improve variable access. It could also special-case `apply` to minimize allocation during primitive procedure application. However, because Scheme functions are mapped in Java method calls on the Java stack, general support for tail recursion cannot be implemented.

2.2 Statement Interpreter

The next logical step would be to take expression analysis a step further to create a statement interpreter. This interpreter would be at the register-machine level with different subclasses of a Java class `Statement` providing the instruction set of the machine. This explicit control over the stack would bring back the possibility of tail recursion. However, there is still a cost of doing a Java method call per `Statement` that is not negligible.

2.3 Byte-code Interpreter

Taking matters to an even lower level of interpretation, the compiler for the statement interpreter could produce its own byte-codes. The byte-code interpreter would be like taking the logic of all the subclasses of `Statement` and merging into one Java method. This would remove the expense of the Java method overhead and instead use the Java virtual machine switch byte-code. Having one large method instead of many small ones also gives the Java just-in-time compiler a better chance to significantly improve the performance of the interpreter. This is similar to the approach taken by Scheme 48, which implements Scheme on top of a byte-code interpreter implemented in the C programming language. [29]

2.4 Byte-code Generation

Instead of implementing a byte-code interpreter in Java, another implementation approach would be to generate byte-codes for the Java virtual machine. [33] This approach is taken by Kawa, another Scheme implemented in Java. [7] However, because this implies use of the Java stack for control flow, it suffers from the same issues regarding tail-recursion as the expression interpreter described above. However, it is possible to do some simple analysis to translate simple tail-recursive loops into regular iteration. This apporach is taken by Pseudoscheme to implement Scheme on top of Common Lisp, with a similar approach used by Kawa. [47] MIT Scheme's

C language backend also uses analysis to cope with an underlying language lacking tail recursive semantics. [14] Some proposals exist for extending the Java virtual machine to support the needs of languages besides Java, but none are available in Sun's reference implementations today. [55]

Chapter 3

First-Pass Implementation

The goal for the first-pass implementation was to get a quick and dirty implementation working with Java 1.0, specifically JDK 1.0.2. Performance and extensibility were not concerns, instead effort was placed into making the implementation multi-threaded and to provide a simple hook-style API from Java into Scheme.

3.1 Beginning

The Scheme implementation described here was originally started as a way of building more experience with Java. Having recently reviewed the second edition of the Structure and Interpretation of Computer Programs, also known as SICP, it was decided to build a little Scheme-like interpreter in Java, perhaps with an Algol syntax. As such, the implementation is based on the SICP chapter-four interpreter. [1]

The implementation later found use in a Java server application with the following requirements:

1. provide customization logic through small code extensions
2. must be able to change customizations without restarting application
3. must be able to interactively test and iterate customizations
4. long-term desire to expose scripting through GUI tool

Scheme's clean language semantics were desired, although there was concern that an s-expression syntax was off-putting to users. Because the original plan was to make a Scheme-like language, and not necessarily a standards-compliant Scheme implementation, the implementation avoided the use of the term Scheme and instead used the terms script and scripting instead of Scheme.

3.2 `Expression.eval`

The implementation revolves around the abstract `Expression` class, with a single `eval` method to implement the logic for each category of `Expression`. The concrete subclasses of `Expression` with their corresponding traditional syntax in the first pass were:

Definition	(define symbol value)
Variable	symbol
Assignment	(set! symbol value)
Quoted	(quote ...)
Begin	(begin ...)
If	(if predicate consequent alternative)
Lambda	(lambda ...)
Application	(...)
Do	(do ...)
Procedure	See analysis

Table 3.1: `Expression` subclasses

The signature of the `Expression.eval` method originally was:

```
abstract public Expression eval (Environment environment);
```

The sole argument to eval is the current `environment`, which is used to evaluate this `Expression`, and passed, possibly modified or extended, when evaluating any sub-`Expressions`. The `Environment` contains an instance field referencing its enclosing `Environment` as well as a static class reference to the `GlobalEnvironment`.

The initial implementations of the `Expressions` were as straightforward as possible to get an implementation working quickly. `Definition` modified the `Environment` by defining a new `Variable`. `Variable` searched through the `Environments` and then the `GlobalEnvironment` to retrieve the value matching its name. `Assignment` performed a similar search through the `Environment` to modify a value. `Quoted` ignored the `Environment`, simply returning its quoted value. `Begin`, `If`, and `Application` pass their `Environment` argument unmodified as they evaluate their sub-`Expressions`. `Lambda` created a new `Compound` `Procedure` in the current `Environment`. `Do` first evaluated its inital values in the current `Environment`, then extended the `Environment` by binding these initial values, and then evaluated its body and condition sub-`Expressions` in the newly extended `Environment`.

The return result of calling `eval` is another `Expression`, possibly and probably a `Procedure` or `SelfEvaluating` `Expression`, which can contain any `java.lang.Object`.

3.3 `Procedure.apply`

`eval` cannot be discussed without its meta-circular companion `apply`. In this implementation `apply` is an abstract method on the abstract class `Procedure`:

```
public abstract Expression apply (Vector arguments)
  throws ScriptException;
```

Because `eval` returned an `Expression` object, `apply` accepts a `Vector` of `Expression` objects. Also for symmetry with `eval`, `Procedure.apply`'s calculated return value is also an `Expression`. [1]

Besides numerous primitive `Procedure` subclasses, there also exists the `Compound` subclass of `Procedure`. As mentioned above, a `Compound` `Procedure` is created by `Lambda.eval`, keeping a pointer to the `Environment` it was created in, as well the `Lambda` `Expression` itself. When `Compound.apply` is invoked, it takes the `Vector` of arguments and uses them to extend its remembered `Environment`, using the variable

[1] In retrospect, `Expression.eval` should take arguments of type `java.lang.Objects` and return a value of type `java.lang.Object`. More on this in section 3.16.1 on page 36 and in section 4.1 on page 41.

bindings stored in the `Lambda` `Expression`. The `apply` method then finishes by evaluating the body sub-`Expressions` of the `Lambda` in this newly extended `Environment`, returning the value of the last sub-`Expression` as the value of the `apply`.

3.4 Syntax

Since no syntax had been decided upon yet, simple programs were constructed in Java, not text files, using `Expressions` subclasses directly for testing the interpreter. `eval` would then be called on the top level `Expression` object.

For example, the Scheme program:

```
;; + is the R5RS function
;; (define + ...)
(define 1+ (lambda (n) (+ n 1)))
(1+ 23)
```

would translate into the Java program:

```
System.out.println(
    new Begin(new Vector(new Object[] {
        new Definition(new Symbol("+"),
                       new Plus()),
        new Definition(new Symbol("1+"),
                       new Lambda(new Vector("n"),
                                  new Application(
                                      new Variable("+"),
                                      new Variable("n"),
                                      new SelfEvaluating(
                                          new Integer(1)))))
        new Application(
            new Variable("1+",
            new SelfEvaluating(
```

```
new Integer(23))))})).eval(new Environment()));
```

The original plan was to avoid the s-expression syntax and instead use something Algol-like to make it more familar to users. This is a familar story for Lisp implementations because even in the early Lisp system the syntax was considered temporary. [39]

JavaCC, the Sun Java Parser generator, provided a first attempt to produce a non-s-expression grammar for scripting. However, after finding that JavaCC could not even parse Java with the official Sun supplied grammar, the effort was abandoned.

3.5 Scheme types and their Java representation

Before continuing in the syntax discussion, note that the above example shows the number one being represented by a `java.lang.Integer`. It is hard to make any progress at this point without nailing down these data-representation issues.

The following table lays out the standard Scheme types and their Java representations:

discriminator	Java class	example
null?	SelfEvaluating	'()
boolean?	java.lang.Boolean	#t #f
symbol?	Symbol	'a
integer?	java.lang.Integer	1
real?	java.lang.Double	1.0
number?	java.lang.Number	1 1.0
char?	java.lang.Character	#\a #\space
string?	java.lang.String or StringBuffer	"string"
pair?	Pair	(cons 1 2)
vector?	java.util.Vector	(vector 1 2 3)
procedure?	Procedure	(lambda ...)
eof-object?	SelfEvaluating	#{EOF}
input-port?	java.io.PushbackInputStream	(open-input-file "file")
outut-port?	java.io.PrintStream	(open-output-file "file")

Table 3.2: Scheme types and their Java representation

3.5.1 SelfEvaluating

`null`, the `eof-object`, and the `unspecified` value are static instances of the `SelfEvaluating` class. `SelfEvaluating` is a simple `Expression` subclass that wraps a `java.lang.Object`. Besides the static instances representing these values, `SelfEvaluating` are allocated to wrap non-`Expression` values that are passed to and returned from `Expression.eval` and `Procedure.apply`

3.5.2 booleans

Originally `booleans` were also implemented as static instances of `SelfEvaluating` but it was quickly realized that for ease of integration with Java, it would be simplest to reuse the `java.lang.Boolean` class. Its two static instances, `Boolean.TRUE` and `Boolean.FALSE` represent Scheme #t and #f respectively.

3.5.3 symbols

The `Symbol` subclass of `Expression` is used to represent Scheme `symbols`. It remembers the name of the `Symbol`, as well as using that for display with `Object.toString`. In that way it is similar to the `SelfEvaluating` class, although they are separate classes so that the `symbol?` will return false for `null` and the `eof-object`.

3.5.4 numbers

As already mentioned, `integers` are represented with `java.lang.Integer`. `real` values are stored using `java.lang.Double`. In general, numbers can be any subclass of `java.lang.Number`, such as `Byte`, `Short`, `Integer`, `Long`, `Float`, and `Double`, or even `java.math.BigDecimal` and `java.math.BigInteger`.

3.5.5 characters

`characters` are represented simply as `java.lang.Character`.

3.5.6 strings

It would seem natural to represent `strings` with `java.lang.String`. However, there is an mismatch between Scheme `strings` and `java.lang.Strings`. The problem is that while Scheme `strings` are mutable, as it true for most conventional programming languages, Java makes a significant departure from most common languages by making `Strings` immutable.

Although Java does have a related class `java.lang.StringBuffer` that does allow mutation, most Java APIs are in terms of the `java.lang.String` class. To make integration easier, it was decided that `string` operations such as `string-length` and `string-ref` would work with both `String` and `StringBuffer` instances, although `string-set!` would signal an error if used with a `String` instance. `make-string` returns instances of `StringBuffer`, so most existing Scheme code dealing with `Strings` will get the behavior they expect.

3.5.7 pairs

`cons` pairs are represented in memory with the `Pair` class which simply contains pointers to two `Expressions`, the `car` and the `cdr`. The `Pair` class includes an implementation of `toString` that correctly handles dotted notation, as well as hiding the dots in list structures.

3.5.8 vectors

`vectors` are represented with `java.util.Vectors`, although an `Object` array would perhaps be more accurate. Unlike Scheme vectors, `java.util.Vectors` are resizable. However, most Java APIs are expressed in terms of `Vectors`, so once again, for interoperability, convenience wins out over exactness. Since Java `Vectors` provide a superset of functionality over Scheme vectors, this should not be problematic.

3.5.9 procedures

As mentioned above, the `Procedure` subclass of `Expression` is used to represent `procedures`.

3.5.10 ports

`input-ports` and `output-ports` are represented with `PushbackInputStream` and `PrintStream` respectively. `PushbackInputStream` provides the necessary functionality to implement `peek-char`, while `PrintStream` can output any `java.lang.Object`, not just `byte` arrays.

3.6 Reader

With Java representations for Scheme `booleans`, `numbers`, `characters`, `strings`, `pairs`, and `vectors` nailed down, it was now possible to write a reader to create them from an `input-port`. The `Reader` class parses s-expressions from any `java.io.InputStream` and returns `Expressions`, which correspond to the the Java representation of any of the aformentioned Scheme types, possibly wrapped in a `SelfEvaluating` `Expression`.

In addition to creating s-expressions, the `Reader` also supports the standard reader macros for `quote`, `quasiquote`, `unquote`, and `unquote-splicing`.

`java.util.StreamTokenizer` provides the basis for the `Reader`, providing simple tokenization and the removing of comments. However, writing a lexer from scratch probably would have been just as easy, in retrospect.

3.7 Expression.analyze

Once the `Reader` was completed, an `analyze` method was added to abstract `Expression` class:

```
public Expression analyze () throws ScriptException
```

`Expression.analyze` would be called on the `Expression` returned from the `Reader` and translate the s-expressions into a program. This method basically performed a type analysis on the `Expression` being analyzed. `Symbols` would be converted into `Variable Expressions`. Non-`Pairs` such as `Strings` and `Numbers` would be converted into `Quoted Expressions`. `Pairs` would be analyzed further based by first recursively analyzing the `car` of the `Pair`. If the resulting `Expression` was not a `Variable`, then that compiled `Expression` is assumed to be the operator of an `Application Expression` and the `cdr` of the `Pair` is compiled to form the operands of the `Expression`.

If the `car` of the `Pair` compiled to a `Variable`, then before the compiler can assume that the `Expression` is an `Application`, the compiler first has to check for special forms. The kernel special-form syntax consists of `define`, `set!`, `quote`, `begin`, `if`, `lambda`, and `do`, which map into `Expressions` as shown in the table above. However, to support the remainder of Scheme syntax, s-expressions are rewritten to transform the special forms `let`, `and`, `or`, and `cond`, into kernel special forms such as `lambda` and `if`. After such rewriting, the new code would in turn be compiled. In the case that the `Variable Expression's` name did not match any special forms, the `Pair` was assumed to represent an `Application Expression` and the operands were compiled as-noted above. When compiling kernel special forms, each special form provided for any necessary compiling of the `cdr` of the `Pair` itself.

3.8 Loader

The next step was to write a `Loader` class. The `Loader` repeatedly calls the `Reader` class. In each iteration it invokes `Expression.analyze` on the result of `read`. It then calls `Expression.eval` on the `Expression` returned and displays the results using `System.out.println`. For the first time the implementation was a working Scheme system that would translate Scheme s-expressions into results.

3.9 Writer

There is a problem with using `System.out.println` to display Scheme values. `println` converts `java.lang.Objects` to `Strings` using the `Object.toString` method. For classes such `Pairs`, `Procedure`, etc., the classes can provide their own implementation `toString` to suit the Scheme behavior, as mentioned above with regard to `Pair`.

`SelfEvaluating` provides an implementation that simply calls `toString` on the `Object` it is wrapping. This works well for printing out `java.lang.Numbers`, since the Java supplied `toString` is what is desired for Scheme as well. It also knows to print the static instances of `null`, the `eof` object, and the `unspecified` as `()`, `#{EOF}`, and `#{unspecified}` respectively.

However, for other Java classes, the standard `toString` behavior does not match what Scheme defines. For example `Booleans`, `Characters`, `Strings`, and `Vector` do not print the way that Scheme users would expect. To provide the expected behavior of the Scheme `write` function, `SelfEvaluating.toString` is extended with additional code to handle displaying `java.*` classes with the expected Scheme semantics. It does this by checking for the known special cases first, such as those mentioned above, using the Java `instanceof` operator and then falling through to use the `toString` in the common case.

A simple `Writer` class bundles `SelfEvaluating`'s ability to convert `Objects` to `Strings` with a `write` method that performs the conversion and then sends the results to an `output-port` implemented as a `PrintStream`.

3.10 Primitives

Although functions like `cons`, `car`, and `cdr` as well as Church numerals could be defined using `lambda` alone, it seems like more practical ways of defining primitives are necessary. This is done by defining primitives in the `GlobalEnvironment` in Java, as was done in the test Java program shown above:

```
Environment.globalEnvironment.define("+", new Plus());
```

Some of the interesting early primitives include `read`, `write`, and `load`, which wrap the `Reader`, `Writer`, and `Loader` classes respectively.

3.11 `Script`

The `Script` class started out as a sort of catch all class. Originally it housed the `SelfEvaluating` instances for values such as `Null`, `EOFObject`, and `Unspecified`. Later it housed the `Script.init` method for defining the primitive `Procedures` as shown above.

Evenually, after a sufficient set of primitives were defined, additional standard functions could be added in Scheme itself. A Scheme file was created to contain these functions. A new method `Script.load` was added to invoke the `Loader`. `Script.init` was extended to load this system initialization file as well.

3.12 `REPL`

At this point, a simple Read Eval Print Loop, or REPL, was written to pull the pieces of the `Script`, `Reader`, and `Expression.analyze`, together into an interactive system. `Script.init` would be called first to initialize the `GlobalEnvironment` and its `Procedures`. Then a `Reader` was initialized on `System.in`. Then the `REPL` class would loop printing a prompt, using the `Reader` to read from `System.in`. If something other than `eof-object` was returned, it would be compiled with `Expression.analyze`. If the compilation suceeded, `Expression.eval` would be called on the returned `Expression`. If the result was other than `Script.Unspecified`, its value would be displayed. Although the `REPL` was not intended to be the interface to this Scheme system, it did provide a great tool for testing and benchmarking.

3.13 `ScriptException`

There have been a couple of references to `ScriptException` in various method signatures and APIs. At this point it seems worthwhile to summarize the common `ScriptExceptions` and their causes:

ArgumentCountException	Procedure.apply	incorrect number of arguments
ArgumentTypeException	Procedure.apply	incorrect type of argument
BoundsException	Vector and String	vector or string index out of bounds
ParseException	Reader and I/O primitives	generally java.io.IOException
ScriptError	Error.apply	allow user functions to signal error
SyntaxException	Expression.analyze	syntax errors
UndefinedVarException	Variable	reading or writing undefined variable

Table 3.3: ScriptException subclasses

3.14 Java-to-Scheme API

The purpose of this implementation is to provide an embedded Scheme language to extend a Java application. To accomplish this, an API is defined for the Java application to interact with the Scheme system. `Script.load` is the first example of such an API.

Loading from a `File` is really just a special case of loading from an `InputStream`. Once there is generalized load from an `InputStream`, a version can be created to load a script from a `String` in memory as well as using a `java.io.ByteArrayInputStream`. This leaves us with three versions of `Script.load`:

- Script.load(InputStream input, String location)
- Script.load(File file)
- Script.load(String script, String location)

The `location` argument is used to identify what is being loaded for error reporting purposes, which defaults to the `File's` name in the `File` case.

`Script.load` is a good starting place, but is not too helpful for integrating Scheme logic into a Java application. Taking an example from Emacs and its use of elisp, the most common form of user extension is the hook. A hook is basically a function defined by the user that is called at a certain point by the application to allow the user to guide the course of execution. The hook function receives a defined set of arguments, and may alter the state of the application through side effects, and perhaps also alter the flow by way of its return value, if the application chooses to use the hook in that manner.

To provide hook functionality, two new methods, `Script.procedure` and `Script.call` were added. `Script.procedure` looks up the value of a `Variable` by name using a third new function, `Script.lookup`, and returns it after making sure the value is in fact a `Procedure`. `Script.call` then allows that `Procedure` to be called with arguments as many times as is desired by the application.

One final requirement is for all of this to work in a multi-threaded environment. Specifically it must be possible for multiple `java.lang.Threads` to simultaneously invoke the `Script` APIs without any danger. Since in this early implementation the only piece of shared state is the `GlobalEnvironment`, basically this comes down to using appropriate Java `synchronize` statements to allow only one `Thread` to modify or access the `GlobalEnvironment` at a time.

3.15 Extensions to Scheme for Java

The last section discussed a Java API for calling Scheme. This section discusses additions to Scheme for accessing parts of Java.

3.15.1 `java.lang.Object`

`java.lang.Object` is a superclass of all Java classes. As such, it contains a number of methods that apply to all Java objects, including therefore the implementation's Scheme objects in Java.

One such method is `Object.toString`. While Scheme provides a selection of `*->string` functions to convert various Scheme types to `strings`, the implementation also provides a more general `to-string` function that converts any Scheme value to a `string`.

Another important method is `Object.equals`. By default `equals` uses pointer equality to compare two Java objects for equality. However, classes can override this simple notion to define class-specific definitions of equality. The most common example of this is the `String` class, which defines equality by comparing the `chars` of each `String` for equality.

Scheme has its own share of definitions of equality including `eq?`, `eqv?`, `equal?`, `char=?`, `string=?`, and `=`. For Java, `equals?` is added to this mix which uses `Object.equals` for comparing two `Objects`. The definition of `equal?` is extended to use `equals?` as a last-resort comparison when testing objects for equality.

Java allows the creation of a new `Object` instance from a `String` class name with the combination of the `Class.forName` and `Class.newInstance` methods. This functionality is provided through the `new` function. This allows us to create many different types of objects besides the ones that the Scheme system knows about out of the box.

Another similar sort of operation commonly found in the Java system is the ability to get and set the fields of an object by name. JavaBeans is one such system, although others exist. However, this general concept of reflection was not available to this implementation because it needed to run in a Java 1.0 environment.

However an alternative was provided for those willing to implement a simple interface. Called `ValueSource`, this interface allows a class to implement a JavaBeans-like protocol through simple `getFieldValue` and `setFieldValue` methods. This functionality is then accessed by `get` and `set` primitive Scheme functions. This allows code to manipulate fields of objects without having to extend the system with primitive `Procedures` for each case, at least for classes willing to implement the `ValueSource` interface. Fortunately, this concept was used extensively in the embedding application to allow a metadata-driven user interface, so it worked out well for the script programmers as well.

3.15.2 java.util.*

The next set of classes to expose to Scheme are the java.util.* utility classes Vector, Hashtable, Enumeration, and Date.

As mentioned before, java.util.Vector provides a superset of what is needed to implement Scheme vectors. To access some of the additions, the extensions vector-addElement, vector-removeElement, and vector-removeAllElements provide access to the Vector methods addElement, removeElement, removeAllElements respectively.

Although Scheme provides a variety of association-list functionality, it is based on list data-structures. Java provides a better performing alternative to association lists, the java.util.Hashtable class. Hashtables can be created with the new function as mentioned above.

At first, Hashtable access was overloaded into the get and set functions mentioned above. This was confusing from a user perspective, since they expected the Hashtable names of get and put instead. It also unnecessarily slowed down get and set because they had to perform an instanceof test to determine if they were dealing with a ValueSource or a Hashtable. Eventually, to avoid the confusion and cost, separate hashtable-get and hashtable-put functions were introduced.

Although hashtable-get and hashtable-put allowed access to individual elements, it did not allow a program to iterate over the keys and elements. To enable this, hashtable-keys and hashtable-elements were added. These return objects of type java.lang.Enumeration. In order to make these return values useful, the functions hasMoreElements and nextElement were added to wrap Enumeration.hasMoreElements and Enumeration.nextElement methods respectively.

java.util.Date objects can be created with the new function mentioned above. A get-time function was added to access the contained numeric value. This was primarily used to compare times when perform benchmarking of the implementation.

3.15.3 Processes

A useful ability of most scripting system is the ability to invoke external commands in sub-processes. Java provides this ability with `java.lang.Runtime.exec`, which creates a `java.lang.Process`. To provide this through Scheme, the implementation provides a simple `exec` function that returns the `Process`. The matching `wait` function takes the `Process` and returns its exit code.

For dealing with the current process, the `exit` function wraps the `System.exit` function, allowing a user to exit the `REPL` process with or without an error code, making it useful for batch operations.

3.15.4 Mail

Another commonly desired ability for scripting systems is sending email. On Unix systems, this can be accomplished by just using the above process machinery to call the standard sendmail program. However, for portability in Java, especially to Win32, a `send-mail` function is provided. This originally was a simple SMTP implementation in Java, but now has been made into a wrapper around the Sun `javax.mail` package.

3.16 Analysis of First-Pass Implementation

Having completed this working first-pass implementation, there are some issues to highlight.

3.16.1 Performance

As mentioned above, `Expression.eval` returns an `Expression`. In retrospect, this return value should have nothing to do with `Expression`, since the tree of `Expressions` represents the static structure of the program, not the run-time values the program produces. This was not just silliness but in fact a serious performance problem, as the cost of allocating wrapper `SelfEvaluating Expressions` and having primitive `Procedures` doing unnecessary and costly `instanceof` operations.

Because `eval` accepted and returned `Expressions`, `Pair`, `Symbol`, and `Procedure` were made subclasses of `Expression`. This avoided having to wrap these classes of objects up in `SelfEvaluating` `Expressions`, but is a symptom of the same problem.

One performance problem that was addressed was the unnecessary use of `Exceptions` for detecting problems. Although Java works hard to make `try-catch` blocks inexpensive when there is nothing to `catch`, using `Exceptions` for control flow does have a cost. Although Java works hard to keep the cost of actually throwing and catching an `Exception` as low as possible, its performance is particularly high when running in the debugger. In many cases, `Exceptions` can be avoided, reserved for truly exceptional conditions.

The first problem along this line was caused by the `String2Number.string2number` method. This method is shared between the `string->number` primitive `Procedure` and the `Reader`. For `string->number` there was not really a problem because it is almost always called by code passing in an actual numeral. However, the performance problem particularly was problematic in the `Reader`. For each `String` token returned by the `StreamTokenizer`, the `Reader` would try to use `string2number` to see if the token was a `number` or a `symbol`. `string2number` first tried to parse the value as an `Integer`, and if that failed, as a `Double`, and if that failed, returned `Boolean.FALSE`. However, the each failure would result in a `NumberFormatException` being thrown and caught.

The reason why this was particularly expensive for the `Reader` is that statistically most tokens are `Symbols`, not `Numbers`. In the `Reader`, `Exceptions` where being used for control flow in the common case, not the exceptional case. The solution was to add some quick tests to guess if the `String` was a `number`. Specifically, if the `String` was empty, or its first character was not a digit or "." or "-", `Boolean.FALSE` was returned immediately. Then if the `String` did not contain ".", it was attempted to be parsed as an `Integer`, while if it did, it would be parsed as a `Double`. Since `Symbols` start with an alphabetic `Character`, `string2number` avoids the `Exception` in the common case, leaving the `Exception` for the truly exceptional case where something that looks like a `number` to the quick test turns out not to be.

The second problem along this line was caused by the Java application calling `Script.load` with a large number of non-existent files. The application intended these files to be optional scripting libraries, so it was not really an error that they were not there. However, the Java run-time was throwing `FileNotFoundException` which the implementation was catching and rethrowing as a `ParseException`. By simply calling `File.exists` before trying to load a file, the application was changed to avoid this cost. This application change ensured the system would start up without any `ScriptExceptions`, greatly increasing startup performance in the debugger.

Finally, on a more positive note, SICP talks about syntax analysis being a performance improvement over the standard s-expression interpreter. [48] Syntax analysis basically is performing the parsing of the text form into the language into data-structures first, and then interpreting that pre-parsed format, instead of reparsing the s-expression on each evaluation. However, because the implementation was in Java and no syntax was defined at the time the core evaluation logic was built, this style fell out naturally by default.

3.16.2 Maintainability

One frustrating limitation of the early implementation is the number of hard coded special cases. The syntax is extensible only from within the implementation of `Expressions`, not via user macros. Similarly, there is no way to add new primitives except through Java.

3.16.3 Standard Compliance

One serious limitiation of this implementation is that it does not support tail recursion. This is primarily because the implementation uses the Java stack for control flow through `Expression.eval`. The current implementation is not a total waste however, because many of the pieces from the primitives to the `Expression` tree could be reused in the future for a different tail-recursive implementation. What is needed is to translate the `Expression` tree into statements similar to the SICP chapter 5 style explicit control evaluator and compiler. As there was no tail recursion, there was no easy way to generally implement `let loop`, and it was omitted.

At this point, a full set of standard library functions was not present. They were added in groups as needed over time. The special function `call-with-current-continuation` was specifically omitted because of the lack of control over the Java stack used to implement `Expression.eval`.

Chapter 4

Second-Pass Implementation

The first-pass implementation was actually employed for some time. It was not complete or well performing but it met the needs of the application using it. More and more primitives and syntax were added to flush out the implementation to more closely approximate standard Scheme. With the amount of effort going into new primitives, work was performed to simplify the writing and addition of new primitives to the system. Performance was improved by simplifying run-time representations and by performing simple compile-time analysis. As the implementation became more widely used, support for debugging the implementation as well as Scheme programs running in the implementation became a new priority.

4.1 Removing `SelfEvaluating` `Expression`

As mentioned above, `Expression.eval` mistakenly returned an `Expression` instead of a `java.lang.Object`. This meant that all `java.*` arguments needed to be wrapped in a `SelfEvaluating` `Expression`.

Because `Expressions` were passed at run-time, primitive `Procedures` expecting non-`Procedure` arguments had to check that the arguments were first `SelfEvaluating` `Expressions`, as well as then checking the type that the `SelfEvaluating` `Expression` contained. Here is an example from `Plus` making sure its argument is a `java.lang.Number`:

```
if (!(object instanceof SelfEvaluating))
```

```
        throw new ArgumentTypeException("Number", object);
    SelfEvaluating se = (SelfEvaluating)object;

    if (!(se.object instanceof Number))
        throw new ArgumentTypeException("Number", se.object);
    Number n = (Number) se.object;
```

In addition, each primitive `Procedure` needed to encapsulate its return value, because as mentioned, `Procedure.apply` returned an `Expression` for symmetry with `Expression.eval`, again an example from `Plus`:

```
    return new SelfEvaluating(new Integer(intResult));
```

In retrospect, the cost of constructing and destructing all of the `SelfEvaluating` `Expressions` seems confusing and expensive. The confusion stemmed from SICP where the Scheme interpreter is written in Scheme. In this system, expressions and s-expression are both represented with `pairs` and other simple values, and although these detail are hidden behind abstraction barriers, apparently that can still cloud the mind of a reader.

With the cleanup of `Expression.eval` and `Procedure.apply`, their method signatures are changed as follows to more natural forms returning `java.lang.Object`:

```
    public abstract Object eval (Environment environment);

    public abstract Object apply (Vector arguments)
      throws ScriptException;
```

4.1.1 `Expression` Inheritance Cleanup

The cleanup of the `Expression.eval` and `Procedure.apply` method signatures enabled various cleanup work in the `Expression` inheritance tree.

`Pair`,`Symbol`, and `Procedure`

It was now clear that is was not meaningful or useful to have `Pair`, `Symbol`, and `Procedure` as subclasses of `Expression`. Now that `eval` and `apply` were cleaned up, these classes were cleaned up as well by simply changing to subclass `java.lang.Object` and by removing their `eval` methods. In addition, the `Pair` class's `car` and `cdr` fields were changed from holding `Expressions` to `java.lang.Objects`.

`Reader`

Now that `Pair` and friends were no longer `Expressions`, the `Reader` had to be changed to return `java.lang.Objects` instead of `Expressions` as well. This meant the Reader could stop wrapping `java.*` values in `SelfEvaluating` `Expressions`.

`Expression.analyze`

Now that the `Reader` returned `Objects`, the `Expression.analyze` method could no longer be an instance method on Expression so it was changed to be a static method instead:

```
public static Expression analyze (Object o) throws ScriptException
```

sub-`Expressions`

Most `Expression` subclasses contain sub-`Expressions`. In the change from `Expression` to `Object` these were also converted. This meant that quoted values no longer had to be boxed with an `Expression`.

However, then `eval` could no longer simply be an instance method on `Expression`. To cope witht his, a static `eval` method was added to `Expression`. It simply checked if the `Object` to evaluate was an instance of `Expression`. If so, it returned `Expression.eval`. Otherwise, it simply returned the `Object` itself to handle the case of quoted values such as `Integers` and `Strings`.

`Constant`

As mentioned before, `null`, the `eof` object, and the `unspecified` object were instances of the `SelfEvaluating` class. A new `Constant` class, a simple subclass of `java.lang.Object`, was created to replace this use of `SelfEvaluating Expression`. Instances of the `Constant` class remember a `String` value to display when `Object.toString` is called. This allows them to display themselves as `()`, `#{EOF}`, and `#{unspecified}` respectively.

`Writer`

The `Writer` class had heavily relied on the implementation of `SelfEvaluating.toString`. Now that `java.*` types were no longer encapsulated in a `SelfEvaluating` object, the logic to print these objects was moved directly to the already static `Writer.write` method.

`SelfEvaluating` **and** `Quoted`

With these changes made, the `SelfEvaluating` and `Quoted` and `Expression` classes were no longer used and they were removed.

4.1.2 Primitive Type Marshalling

Removing `SelfEvaluating Expression` meant visiting all of the primitve `Procedures` to cleanup their argument type handling code. The primitives had largely grown through cut-and-paste, so there was a lot of duplicate code for common argument validation. Where argument parsing code had not been cut-and-paste, subtle differences in behavior had arisen in some cases.

Since all primitives were being revisited, a set of helper functions was created. The `Script` class took on this new type-marshalling role.

`Script.string` was the first such method introduced. It handled automatic conversion of `StringBuffers` used to represent mutable Scheme `strings` into immutable Java `Strings` as needed for interfacing with Java code.

This was soon followed by `Script.object`, which handled converting from the `Constant Script.Null` to the Java `null` value, as well as possibly converting `StringBuffers` to `Strings`. `Script.object` would be used by any code such as `hashtable-put` that received a `java.lang.Object`, where the implementation would want to convert from its representations into something more expected for Java code.

Although both `Script.object` and `Script.string` could convert `StringBuffers` to `Strings`, there are differences. Basically, `Script.string` would raise an `ArgumentTypeException` if it did not receive a `String` or `StringBuffer`. In `Script.object`, the conversion was done if appropriate, but any other values would pass through without raising any `ArgumentTypeException`.

As time went on, many type-marshalling methods were added to `Script` to deal with all the common types, from `Number` to to `Vector` to `Hashtable` to `Enumeration` to `Date`, etc. All of these marshalling functions throw an `ArgumentTypeException` if the expected type is not passed and not derivable from the type passed, such as converting a `StringBuffer` to a `String`, with the as-noted `Script.object` which can handle any value.

These type-marshalling methods simplifed all of the primitives greatly because all of error handling for most functions moved to helper methods. This made the Java code have a more functional style and improved readabilty. It also made it easier for programmers to add new primitives by allowing the primitives to focus on their specific task, and not on the Scheme representation details.

4.2 Compiler

Some of the biggest changes in the second pass revolved around the new `Compiler` class. `Expression.analyze` was moved out of `Expression` to form `Compiler.compile` which was then enhanced.

4.2.1 `CompileTimeEnvironment`

The first change was to introduce `CompileTimeEnvironments`. By using `CompileTimeEnvironments` the `Compiler` can take advantage of lexical scoping to change run-time searching of the `Environment` into a compile-time search of a `CompileTimeEnvironment`. Therefore, as part of the move from `Expression.analyze` to `Compiler.compile`, a new `CompileTimeEnvironment` argument was added, resulting in the following signature:

```
public static Expression compile (
    Object                 object,
    CompileTimeEnvironment environment)
  throws ScriptException
```

The `CompileTimeEnvironment` argument is extended with a new `CompileTimeEnvironment` whenever a `lambda` special form is compiled. The extended `CompileTimeEnvironment` remembers the variables bound by the `Lambda Expression`.

In order to take advantage of the `CompileTimeEnvironment` information, it is necessary to replace the `Variable` and `Assignment Expression` classes. `Variables` that are found in the `CompileTimeEnvironment` are represented with `LexicalAddress Expressions`, while those that are not found are represented with `GlobalVariable Expressions`. `Assignments` are represented with `LexicalAssignment` and `GlobalAssignment` respectively. The new classes are summarized in the following table:

GlobalVariable	symbol
GlobalAssignment	(set! symbol value)
LexicalAddress	symbol
LexicalAssignment	(set! symbol value)

Table 4.1: Variable and Assignment replacement Expression subclasses

4.2.2 `GlobalVariables` as Cells

The `GlobalEnvironment's` implementation started out using a `Hashtable` mapping `String` variable names to values. `GlobalVariable.eval` and `GlobalAssignment.eval` were implemented with `Hashtable.get` and `Hashtable.put`.

While a `Hashtable` lookup is usually constant time, as mentioned before all access to the `GlobalEnvironment's Hashtable` had to be `synchronized` to ensure safe multi-threaded access. This meant that the `GlobalEnvironment` had become a bottleneck.

To solve this, the `GlobalEnvironment's Hashtable` was converted from storing values to storing cells. The cell contains the value of the variable, and `GlobalVariable` and `GlobalAssignment` references the cell, reducing the cost of access to a field reference and assignment. Instances of the previously static `GlobalVariable` class are used to represent the cells.

By simply changing `GlobalVariable` access from a `synchronized Hashtable` access to a compile-time `Hashtable` lookup with a run-time field reference, the performance of `(fib 30)` improved by 25%.

`Symbol`

While changing `GlobalVariables` into cells, it was discovered that the `Symbol` class redefined `Object.equals` to using `String.equals` instead of simple pointer equality. Apparently this dated back to the early Java test days when `new Symbol` was used when writing test programs, as shown above in section 3.4 on page 24. `Symbols` should be interned, so that if two symbols have the same name, they should be the same object, that is, pointer equals.

To fix this, the `Symbol` constructor was made `private`, and a new `Symbol.get` method was added. `Symbol.get` creates a new `Symbol` for a name only if one does not exist, otherwise it returns the existing `Symbol`. Since this method uses a global symbol table, it needs to be `synchronized` to prevent safe multi-threaded access.

Although the Scheme standard references the concept of an uninterned symbol, it is not required and is not supported by this implementation.

4.2.3 Table-Driven Syntax

In the older `Expression.analyze`, `Pairs` were compiled by compiling the `car`, and if it was a `Variable`, then checking the `Variable` name exhaustively for both kernel special forms and syntax that needed to be rewritten into kernel special forms. In the move to `Compile.compile`, this was changed to handle syntax in an extensible manner.

The new `GlobalVariable` cells were extended with a type field, which has the possible values of `Location`, `Special`, or `Macro`. `Location` indicates that the `GlobalVariable` is simply a traditional location containing a value. `Special` indicates that the `GlobalVariable` is holding a `SpecialFormCompiler`. Finally, `Macro` indicates that the `GlobalVariable` is holding a rewriter `Procedure`.

So now, instead of exhaustively searching to see if the name matches a special form or syntax to rewrite, when the compiler compiles the operator position of a `Pair` to a `GlobalVariable`, it simply looks at the `GlobalVariable's` type field to decide what to do next, the details of which follow.

Special Forms

The first case the compiler checks for is `Special GlobalVariables`. If one is found, `Compiler.compile` passes the `Pair` and `CompileTimeEnvironment` to the `SpecialFormCompiler` contained in the `GlobalVariable's` cell. `SpecialFormCompiler` is an interface with one method:

```
public Expression compileSpecial (
    Pair                   pair,
    CompileTimeEnvironment environment)
  throws ScriptException
```

This is basically the same signature as the `Compiler.compile` method, although in this case the compiler has already determined that it is compiling a `Pair` and not just any `java.lang.Object`. The `SpecialFormCompiler` throws `ScriptException`, usually to indicate that `SyntaxException` has occured.

The old special-form code from `Expression.analyze` was moved to several new `SpecialFormCompilers` classes. These `SpecialFormCompilers` are registered by the new `Compiler.init` method, which plays a similar role to `Script.init`. In this case it initializes the `SpecialFormCompilers`, as opposed to primitive `Procedures`, into the `GlobalEnvironment`.

Macros

The second case the compiler checks for is `Macro GlobalVariables`. If one is found, the compiler creates and evaluates an `Application Expression`, using the value of the `GlobalVariable's` cell as a `Procedure`, and the `Pair` as the sole argument. The resulting s-expression is then compiled in place of the original.

The syntax `let`, `cond`, `or`, and `and` are handled as `Macros` with rewriter `Procedures` defined in Java. However, a new primitive function `define-rewriter` was added which takes a symbol and a function, defining not a normal `Location GlobalVariable`, but a `Macro GlobalVariable`. This allows user-defined syntax. For example, `let*`, `quasiquote`, `case`, `letrec`, and `delay` are all defined using `define-rewriter`. The simplest example is `delay`:

```
;; promises
(define make-promise
  (lambda (proc)
    (let ((result-ready? #f)
          (result #f))
      (lambda ()
        (if result-ready?
            result
            (let ((x (proc)))
              (if result-ready?
                  result
                  (begin (set! result-ready? #t)
                         (set! result x)
```

```
                        result))))))))

  (define-rewriter 'delay
    (lambda (expr)
      (list 'make-promise `(lambda () . ,(cdr expr)))))

  (define (force promise)
    (promise))
```

A `gensym` function was added to generate unique symbols for use in `define-rewriter` macros. `gensym` created symbols beginning with `---` so they will not conflict with symbols that might appear in Scheme program source. An `error` function was added to allow macro writers to signal their own syntax errors from macro rewriter functions.

Locations

Finally, if the `GlobalVariable` is not `Special` or `Macro`, then it is simply a `Location`, and the `Pair` is compiled as an `Application`.

NaryLambda

As part of the `Compiler` work, as a prerequisite to filling in missing standard scheme functions, support for n-ary arguments was added. This was done within the scope of the `lambda` `SpecialFormCompiler`. `NaryLambda` was added as a subclass of `Lambda`. It differs in that it creates an `NaryCompound` when `eval` is called as well in that it overrides `toString` to handle the correct printing of the argument list. `NaryCompound` is a subclass of `Compound` that handles the allocation of the list from any optional arguments passed to `apply`.

4.3 Primitives

With a working language implementation from the first pass, a lot of the effort in the second pass went to filling out missing primitives and cleaning up existing ones, as well as the mechanisms supporting them.

4.3.1 I/O Primitives

Several Scheme I/O functions rely on a default value for their input or output port, for example, read and write. The functions `with-input-from-file` and `with-output-to-file` can change this default value during the execution of a thunk.

In a multi-threaded Scheme implementation, a simple global value cannot be used to track the current value of these default ports. This implementation uses thread-local storage to track the defaults per thread using the primitives `hashthread-state`. The String keys `current-input-port`, `current-output-port`, and `current-error-port` are used to track the different default ports uniquely for each thread.

Although in later Java APIs thread local storage is provided, in Java 1.0 and Java 1.1 an implementation has to provide its own. For this implementation, `Thread.currentThread` is used to index into a `Hashtable` that maps from `Threads` to a `Hashtable` of thread local state. The inner `Hashtable` maps from `String` keys in the various values. Two primitive `Procedures` `set-hashthread-state` and `hashthread-state` allow access to these values from Scheme.

One problem with this simple Java 1.0 implemention is there is no general way to garbage collect the thread-local storage when a `Thread` exits. One approach is to override `Thread.run` or provide a wrapper `Runnable` to cleannup the thread-local storage when the `Thread` exits. One improvement in Java 1.1 is the ability to use `java.lang.ref.WeakReferences` to create a hastable that will allow the `Threads` keys values to be garbage collected.

In addition to thread-local storage, the `dynamic-wind` function was added and used to implement `with-input-from-file` and `with-output-to-file`. The implementation of these functions utilizes `dynamic-wind` with a begin thunk that uses `hashthread-state` to remember the old value and `set-hashthread-state` to set the new value followed by an after thunk that then restores the old value using `set-hashthread-state`.

`transcript-on` and `transcript-off` also require some special support in manipulating the default ports. When a transcript is turned on, any output to the `current-input-port`, or the implementation extension `current-error-port`, needs to be redirected to the transcript file. To do this, the `PrintStream` used to represent the output ports is replaced with a special `MultiPrintStream`. The `MultiPrintStream` subclass of `PrintStream` multiplexes output methods over several `PrintStreams`. This allows output to be automatically sent both to the normally intended destination as well as the transcript without having to change the I/O primitives to be aware of the new transcript functionality. However, the `REPL` class was changed to be aware of the transcript so that if a transcript is on, the interactively input expression is sent to the transcript as well as the resulting value.

4.3.2 Externalizing Primitive Definitions

As mentioned before, added primitive functions are registered by `Script.init`, which means adding new primitives requires changing Java code and recompiling. In addition, as mentioned, `Compiler.init` defines rewriters for some special forms in the similar hard-coded way. This is problematic because it prevents application users from easily extending the system with their own primitives without modifying the interpreter sources.

However, the Java `new` extension function already presents a tidy solution to this problem. As mentioned before, the `new` function takes a Java `String` class name and creates an instance of that class. Since the primitives are simply Java classes, this means the implementation can use `new` to define all of the primitives in Scheme itself, with the special exception of the `new` primitive itself. The result looks like this:

```
(define eq? (new "Eq"))
```

This also lets us remove the similar code in `Compiler.init` as well, since first `define-rewriter` can be defined and then `define-rewriter` can be used to register the new sytax:

```
;; Syntax extension
(define define-rewriter (new "DefineRewriter"))
```

```
(define gensym          (new "GenSym"))
(define-rewriter 'let   (new "Let2Application"))
(define-rewriter 'cond  (new "Cond2If"))
(define-rewriter 'or    (new "Or2If"))
(define-rewriter 'and   (new "And2If"))
```

4.3.3 Removing Non-Primitive Primitives

Cleaning up the hard-coded primitives from Java showed that there was a lot of unnecessary Java code in the initialization of the system. Further inspection of the existing primitives shows there are was a lot more unnecessary Java code in the implementation of the primitives.

Some simple examples of unnecessary Java code are classes like `NullP` and `BooleanP` which can be replaced with Scheme code such as:

```
(define (null? x) (eq? x '()))
(define (boolean? x) (or (eq? x #t) (eq? x #f)))
```

In other cases, adding one new Java primitive can obsolete many others. A new `instanceof?` primitive function allows access to the Java `instanceof` operator from Scheme. By using it, the system can leverage the knowledge of Java implementation to reduce the number of Java primitives. For example, all of the type discriminators were replaced as follows:

```
(define (pair? x)        (instanceof? x "Pair"))
(define (symbol? x)      (instanceof? x "Symbol"))
(define (procedure? x)   (instanceof? x "Procedure"))
(define (vector? x)      (instanceof? x "java.util.Vector"))
(define (input-port? x)  (instanceof? x "java.io.PushbackInputStream"))
(define (output-port? x) (instanceof? x "java.io.PrintStream"))
(define (char? x)        (instanceof? x "java.lang.Character"))
(define (number? x)      (instanceof? x "java.lang.Number"))
(define (real? x)        (instanceof? x "java.lang.Double"))
```

```
(define (integer? x)     (instanceof? x "java.lang.Integer"))
(define (string? x)  (or (instanceof? x "java.lang.String")
                         (instanceof? x "java.lang.StringBuffer")))
```

Even access to certain magic values such as `Script.Unspecified` can be created from existing code:

```
(define (unspecific) (if #f #f))
```

The unspecific function was useful when defining standard functions or macros that are supposed to return an unspecific value, without having them return some arbitrary value or adding a primitive just to access it from Java.

4.3.4 Partitioning Primitive Definitions

Now that the primitive `Procedure` definition has been externalized and minimized, it is beneficial to put in place some additional structure. This is done by splitting the primitives and other definitions that have accumulated into three categories: standard Scheme, Java extensions, and application extensions.

The three categories are split into three separate Scheme files: `system.scm`, `util.scm`, and `application.scm`. Now an interpreter can choose what set of definitions to provide. The `REPL` used for testing for example only loads the `system.scm` and `util.scm` definitions. The embedding application can choose to load its own extensions with `Script.load` after it calls `Script.init`.

4.4 Arrays

One general representational change in the second-pass implementation was to switch to use Java arrays in place of other higher level data structures. Although these can require more work to use in general, they do provide performance benefits. One performance benefit is lowered memory usage. Usually a higher level data-structure is just a wrapper around an array, so using the array on its own removes the encapsulating object. Another performance benefit is faster access. Using a wrapper object places read and write access, even length access, behind the extra cost of a method call. Using an array directly removes these extra costs. Additionally, higher level data-structures may also provide unnecessary synchronization overhead when objects are used within a single thread.

4.4.1 `StringBuffer` to `char[]`

One major representation change was to change the Java representation of Scheme strings from `StringBuffers` to `char[]`. This turned out to be quite easy in fact, thanks to the conversion to using `Script.string`. A few rare places did need to treat `String` and `char[]` separately but were easily found because they were the same places the code used to special case `StringBuffer`.

As discussed above, `StringBuffer` was originally chosen because `Strings` are immutable. However, `StringBuffers` have additional functionality such as the ability to grow which is not needed to implement Scheme `string` semantics. In addition, all operations on `StringBuffers` are `synchronized`, which does have a cost, even when the `StringBuffer` is not shared between `Threads`.

4.4.2 Arguments from `Vector` to `Object[]`

In addition, a number of internal `Vectors` where changed to use `Object[]`. The most visible place for this was in `Procedure.apply`, which changed to this form:

```
public Object apply (Object[] arguments)
  throws ScriptException
```

`Vector` remained as the Java representation for Scheme `vectors` for ease of integration, but internally in most cases its resizability and implicit synchronization were not needed.

4.5 `Application` Special Cases

When changing `Procedure.apply` to take a `Object[]` instead of a `Vector`, it became clear that it would be better if it did not have to take even an `Object[]`. For example, the `Cons Procedure` should be able to get its two argments without allocating an argument array to hold them.

This is in fact a relatively easy change conceptually, although it does mean changing all the primitive `Procedures` in mechanical ways. First the `Procedure` class is changed not just have one `apply` method, but several, corresponding to different numbers of arguments:

```
abstract public Object apply0 ()
  throws Exception;
abstract public Object apply1 (Object o1)
  throws ScriptException;
abstract public Object apply2 (Object o1, Object o2)
  throws ScriptException;
abstract public Object apply3 (Object o1, Object o2, Object o3)
  throws ScriptException;
abstract public Object apply4 (Object o1, Object o2, Object o3, Object o4)
  throws ScriptException;
abstract public Object applyN (Object[] objects)
  throws ScriptException;
```

Then convenience subclasses `Procedure0`, `Procedure1`, `Procedure2`, `Procedure3`, `Procedure4`, and `ProcedureN` are provided for primitives to use. `Procedure2` looks like this:

```
public abstract class Procedure2 extends Procedure {
    public Object apply0 ()
      throws ScriptException {
        throw new ArgumentCountException(2, 0);}
    public Object apply1 (Object o1)
      throws ScriptException {
        throw new ArgumentCountException(2, 1);}
    public abstract Object apply2 (Object o1, Object o2)
      throws ScriptException;
    public Object apply3 (Object o1, Object o2, Object o3)
      throws ScriptException {
        throw new ArgumentCountException(2, 3);}
    public Object apply4 (Object o1, Object o2, Object o3, Object o4)
      throws ScriptException {
        throw new ArgumentCountException(2, 4);}
```

```
    public Object applyN (Object objects[])
      throws ScriptException {
        throw new ArgumentCountException(2, objects.length);}}
```

This reduces the `Cons Procedure` down to the simple and efficient:

```
public class Cons extends Procedure2 {
    public Object apply2 (Object o1, Object o2) {
        return new Pair(o1, o2);}}
```

Most primitives now have no argument `count` checking at all, since it is implied by their superclass. However, some classes are not so simple, and for them `ProcedureN` is provided. It is used for functions that can take more than 4 arguments, such as `send-mail`, or a varying number of arguments and want to share one `applyN` method, such as `+` and `-`. To facilitate this, the `apply0`, `apply1`, `apply2`, `apply3`, and `apply4` methods of `ProcedureN` simply package up their arguments in an array and call `applyN`:

```
public abstract class ProcedureN extends Procedure {
    public Object apply0 ()
      throws ScriptException {
        return applyN(new Object[] {};)}
    public Object apply1 (Object o1)
      throws ScriptException {
        return applyN(new Object[] {o1});}
    public Object apply2 (Object o1, Object o2)
      throws ScriptException {
        return applyN(new Object[] {o1, o2});}
    public Object apply3 (Object o1, Object o2, Object o3)
      throws ScriptException {
        return applyN(new Object[] {o1, o2, o3});}
    public Object apply4 (Object o1, Object o2, Object o3, Object o4)
      throws ScriptException {
        return applyN(new Object[] {o1, o2, o3, o4});}
    abstract public Object applyN (Object objects[])
      throws ScriptException;}
```

In addition, if a `Procedure` can take a variable number of arguments, such as `read`, additional `apply` methods can be overriden, instead of just the abstract one, without resorting to `ProcedureN`. In addition one `apply` method can call another, as in the case of `read` where `apply0` can call `apply1` with the defaulted `input-port` argument.

However, simply changing `Procedure` and its subclasses is not enough. The `Application Expression` class which called `Procedure.apply` needs to be expanded into `Application0`, `Application1`, `Application2`, `Application3`, `Application4`, and `ApplicationN` which each call their respective `apply` method.

As mentioned, the `Compiler` creates `Application Expressions` when compiling a `Pair` that is not a special form or a macro. The new `Compiler.makeApplication` method now analyzes the argument list to the application to decide which type of `Application Expression` to create. In addition to the `Compiler` itself, the `apply` primitive and `Script.call` API are also changed to use `Compiler.makeApplication`, so they can create the correct `Application` object at run-time.

4.5.1 Unrolling Primitives

In order to further cut down on unnecessary allocations in argument passing, something more can be done about subclasses of `ProcedureN`, which still pass their arguments in an `Object[]`. Usually subclasses of `ProcedureN` are for primitive functions with an unlimited number of arguments such as such as `apply`, `=`, `<`, `-`, `+`, `*`.

These primitives are structured with an internal loop to handle the arbitrary number of arguments. However, in most cases, they are called with a small number of arguments, usually within the bounds of our `Application` special cases for zero to four arguments. To take advantage of this, the loop provided for the `applyN` case can be unrolled, specializing it for smaller numbers of arguments. For values that are too small to be legal, a method can be overriden to throw an `ArgumentCountException` as `Procedure2` demonstrated above.

By adding these special versions of primitives, the time to run `(fib 30)` by reduced by 33%.

4.6 Handling of `Exceptions`

The implementation tries to protect the caller of `Script.eval` from any `Exceptions` arrising out of executing a possible user supplied script. However, in practice it is not practical do this, and sometimes it is not even desired.

Java exceptions are really all subclasses of `java.lang.Throwable`. `Throwable` in turn is partitioned in subclasses of `java.lang.Error` and `java.lang.Exception`. In general, `Errors` should not be caught, and including things such `java.lang.LinkageErrors` resulting from class files that are corrupt, such as through truncation, or invalid, such as those with circular inheritance hierarchies.

Futhermore, `java.lang.Exception` is partitioned, albeit less symmetrically, into classes that are subclasses of `java.lang.RuntimeException`, and those that are not. `RuntimeExceptions` include common programming errors such as `NullPointerExceptions` and `ClassCastExceptions`. Non-`RuntimeExceptions` are `Exceptions` that are explictly declared by a method. Since the implementation has control over the signature of `Expression.eval`, it knows that the only non-`RuntimeException` thrown is its own `ScriptException`.

However, sometimes a `RuntimeException` may not be the fault of the script itself, and should not be surpressed. An example of this is in a transactional system were a deadlock has been detected and a higher level part of the system may want to retry the transaction after first rolling back. To handle this case, the `Script` class allows the registration of certain classes of `RuntimeExceptions` that are to be rethrown automatically if they are encountered, to allow higher level handling to run. The signature of `Script.eval` does not need to change because it is not necessary to declare the rethrowing of `RuntimeExceptions`.

4.7 Debugging

Debugging features are not part of the language standard and as such usually get little attention and poor support. One type of debugging was needed to aid in the implementation of the new compiler features. In addition, as the focus shifted from work on the interpreter to actually using the interpreter, there was a need to aid programmers in debugging their Scheme code.

4.7.1 Java Debugger

As mentioned, one type of debugging is debugging the interpreter itself. Most debugging of the interpreter was done using the standard Java `jdb` debugger.

When inspecting run-time data-structures, `jdb` allows `Objects` to be inspected with the two commands `dump` and `print`. The `dump` command displays each field of an object in a standard format, but is not good for getting a high level view of a data strucutre. For example, a `Hashtable` is displayed as two parallel `Object[]` along with other fields for the usage and size etc., not a mapping from keys to values. However, the `print` command uses the `Object.toString` method to render the `Object` for display, resulting in a usually more useful presentation of information. For example, a `Hashtable`is displayed as a simple text table showing the mapping of keys to values. To improve debugging within the `jdb` debugger, it is therefore important to provide useful `Object.toString` implementations for the implementations various classes.

Run-Time Values

Table 3.2 on 25 provided a list of the system's various run-time values. For the Java classes `Boolean`, `Integer`, `Double`, `Number`, `Character`, `String`, `Vector`, `PushbackInputStream`, and `PrintStream`, the system already provides a reasonable `Object.toString` implementation. [1] In the discussion of the `Constant`, `Symbol`, and `Pair` classes, it was mentioned that an `Object.toString` method was defined to provide a useful display representation.

That leaves the `Procedure` class as the one class that does not have a `toString` implementation. An `Object.toString` method could be added to each of the approximately one hundred primitive `Procedures` in the system, but that would mean duplicating the names of functions both in the Scheme file that defines them and in the implementation of the `Procedures` themselves.

Instead, a `Symbol name` field was added to the `Procedure` class. The `Definition Expression` was changed so that when a top level define is evaluated, it checks to see if the value is a `Procedure`, and if so, stores the `Symbol` being defined in the `Procedure's name` field. Then the new `Procedure.toString` method can include the name of the `Procedure`

[1] Remember that the Writer class does exists to display many of these Java objects in their correct Scheme form, however the default `toString` is good enough for use in `jdb`.

if one is available, or the default `Object.toString` if one is not. One might not be available if the `Procedure` is anonymous or was assigned to a global variable with `set!` instead of with `define`. The run-time cost of this mechanism is low, because global variables are usually only defined once and afterwords are usually changed with `set!`.

After this, all of the `Script` type marshalling code that could cause ArgumentTypeExceptions, such as `Script.string` and `Script.pair` etc., were changed to take a `Procedure` argument. The implementation of these type marshallers could then let programmers know not only that they had passed an `integer` where was a `pair` was expected, but also that the procedure expecting the `pair` was named `car`. In addition, `ArgumentCountException` was extended to take a `Procedure` as well for a similar usability improvement.

`Expression` values

In addition to providing `Object.toString` for run-time data values, the `Expression` classes also need to be inspected in the debugger. Although, as mentioned above, `Expression` only defined one abstract `eval` method for subclasses to override, it is now convenient to also have each override `Object.toString`. For example, the `If` class would display as `(if ... ... ...)`, in effect reversing the compilation.

One problem is with the introduction of `LexicalAddress` with `CompileTimeEnvironment` and `GlobalVariable` cells, variables no longer remembered their `Symbol` name since it was no longer necessary at run-time. However, in order to provide debugging, these specific `Expressions` needed to be changed to store more symbolic information for debugging. The `Compiler` can then store that information into the `Expressions` as it creates them.

Even `Lambdas` do not need to remember the names of the variables they bind, and likewise the run-time `Environment` class no longer knows the names of the variables stored within it. However, to make both of these more useful for debugging, the `Compiler` was changed to store this information in `Lambda`, and the `Compound Procedure` passes this information when it extends the `Environment`.

4.7.2 Stack Traces

While these debugging changes had some positive impact on the Scheme developer, they were targeted primarily at the Scheme implementor. While it is helpful to know that `cons` was called with the wrong number of arguments, a program might call `cons` in a lot of places. A programmer needs to know the context for any given error. One form of context that is useful is a stack trace showing the currently pending computations.

Providing a stack trace turned out to be relatively easy. As mentioned above, all `Expression` evaluation is now funneled through a static `Expression.eval` method. A Java `try/catch` block was put around the static `eval` method's invocation of the instance `eval` method. The `catch` block would catch any `ScriptExceptions` that were thrown. It would handle the `ScriptException` by printing out the `Expression` being evaluated using the `Expression.toString` discussed earlier and then rethrow the `ScriptException`.

When a `ScriptException` occured, the Java stack would unwind the call to the static `Expression.eval` method, printing the `Expression` that was being evalutated, and then rethrowing the `ScriptException` to the next level. At the top level, `Script.eval` would then print the `ScriptException` itself. The output then contained both the error as well as the context that the error occured in.

One additional detail is the handling of `RuntimeException`. `Script.eval` used to handle these `RuntimeExceptions` at the top level to prevent them from escaping to the calling program. However, now a `RuntimeException` would pass through the stack trace machinery without providing the context information. To fix this, the `Application` `Expression` classes were changed to catch the `RuntimeException` and convert it into a new `ScriptException` subclass, `PrimitiveException`. Then when `Application` catches the `RuntimeException` and throws its `PrimitiveException`, the stack trace machinery will behave properly.

4.7.3 Source

One problem with the stack trace mechanism is its use of `Expression.toString`. While the Scheme implementor might be happy to see code in terms close to its internal kernel representation, programmers would prefer to see their code the way they wrote it using syntactic sugar.

In order to provide this programmer context, the `Expression` class changed to optionally remember the `Pair` it was compiled from. Then the stack trace can display the users code instead of the internal representation when it is available.

Sometimes a user might not know the location of the code even if shown the source, perhaps because the system is large or is a collaboration between multiple users. In order to provide file and line number information for source code, a new `DebugPair` class was added. `DebugPair` is a subclass of `Pair` that can remember the source location for a `Pair`. The `Reader` was changed to create `DebugPair's` when loading source code. `Expression.eval` can then including the location of the source for the stack trace.

This simple implementation has the unfortunate side effect of retaining the full source code in memory as s-expressions. Traditionally, systems just remember the location of the source and then read it in from the original location as needed for reporting errors. One problem with the traditional method is that it usually depends on the fact that all source comes from the file system, which is not necessarily true in an embedded system. In practice, the extra memory has not been a concern, since it is only allocated once at compile-time and not repeatedly at run-time.

4.7.4 `REPLServer`

One more interesting debugging feature was the `REPLServer`. The `REPL` class was cleaned up to not include initialization of `Script` and the various dependencies on the standard Java streams `System.in`, `System.out`, and `System.err` were factored out. This allowed multiple `REPLs` to run simultaneously. Now a simple service was created to allow telnet access to the running application, which would run a `REPL` using the existing initalized `Script` state, performing interaction over the network connection instead of the console. This allowed testing of code and inspection of the application state from outside the application.

4.8 Analysis of Second-Pass Implementation

After the second-pass implementation, some issues have been resolved. `Objects` are passed around at run-time, not `Objects` wrapped with `SelfEvaluating Expressions`. The implementation is easier to maintain and extend with the externalization of initialization of primitive functions and syntax rewriters. The `Compiler` was introduced, with the resulting compile-time analysis resulting in improve run-time performance. However, even with these improvements, there are still issues to discuss.

4.8.1 Modules

The attempt to separate the implementation into `system.scm` and `util.scm` was not as clean as one would like. The goal was to place only standard R5RS definitions into `system.scm` and place all the non-standard extensions into `util.scm`. However, because some of the `system.scm` implementations depended on the `util.scm` extensions this clean split was not possible. Some examples of `util.scm` functions needed by `system.scm` mentioned previously are `new`, `instanceof?`, `define-rewriter`, `gensym`, and `error`.

What is needed is a module system. This would allow the system to be built upon non-standard internals, but not necessarily expose them in the environment. Then a programmer could choose from a standard Scheme environment or optionally import modules providing specific extensions.

4.8.2 Performance

There were a number of performance issues related to the second-pass implementation.

`Symbol` Performance

The `Symbol` performance problem was a surprise to find in hindsight. However, it does not seem that such poor performance is not standards-compliant. According to R5RS, section 6.3.3 Symbols:

> Symbols are objects whose usefulness rests on the fact that two symbols are identical (in the sense of eqv?) if and only if their names are spelled the same way.

Similarly, R5RS section 6.1, Equivalence predicates, defines `eqv?` in terms of `string=?` and `symbol->string`:

> The eqv? procedure returns #t if:
>
> - ...
> - obj1 and obj2 are both symbols and
>
> ```
> (string=? (symbol->string obj1)
>           (symbol->string obj2))
> ```

`$\Longrightarrow$ \#t`

- ...

So in fact, the implementation is standards-compliant. However, later in section 6.1:

> Eq? and eqv? are guaranteed to have the same behavior on symbols...

So first `eqv?` is defined in terms of `string=?` and then later `eq?` on symbols is defined to be the same as `eqv?`. Finally, the discussion sheds some light:

> Rationale: It will usually be possible to implement eq? much more efficiently than eqv?, for example, as a simple pointer comparison instead of as some more complicated operation.

In fact, looking back at the original implementation of `Eq` primitive `Procedure` before the `Symbol` interning change, it did in fact special case `Symbol` equality as defined in section 6.1. After the interning change, `Eq` was cleaned up to follow the intent in the Rationale.

I/O Performance

As mentioned before, the implementation uses `java.io.PushbackInputStream` and `java.io.PrintStream` to represent `input-ports` and `output-ports` respectively. However, using these directly without using underlying `java.io.BufferedInputStreams` and `java.io.BufferedOutputStreams` meant suffering with character-at-a-time input and output. Once again, this was easy to correct once known. Most standard language libraries specify the details of buffering, but once again the Scheme standard does not address the subject.

Buffering is not just a performance issue but is also affects the writing of programs. For example, this simple script approximates the `REPL`. Note the `(newline)` after the `display` of the prompt. Now a `flush` function is needed to flush the buffering at arbitrary points.

```
(define (repl)
  (let ((prompt (lambda ()
                  (display "> ")
                  (newline)       ; (newline) to force flush of prompt
                  (read))))
    (do ((s-expression (prompt) (prompt)))
```

```
        ((eof-object? s-expression) (exit))
      (write (eval s-expression (interaction-environment)))
      (newline))))
(repl)
```

4.8.3 Macros

The `define-rewriter` macros are similar to most non-standard macro extensions. The R5RS standard with macros was not published when the implementation reached this point so they were not implemented.

R5RS macros are a much better approach besides the resolution of namespace issues since they are much easier to write. A major problem encountered with the `define-rewriter` macros is that it is too easy to tolerate unexpected syntax by not properly checking the structure of s-expressions. Code that manually parses s-expressions often overlooks error cases. Even the internal `if SpecialFormCompiler` had a problem of accidentally tolerating the illegal `(if 1 2 3 4)`.

At the very least `define-rewriter` should probably be made source compatible with Lisp style `defmacro` syntax. The world does not need yet another macro system.

Chapter 5

Third-Pass Implementation

The second-pass implementation was the first deployed in an application. Additional performance analysis on the overall application pointed out some additional performance issues in the implementation resulting in another pass over the implementation.

Even with the optimizations for `Procedure[01234N]` and `Application[01234N]`, the implementation still allocates a significant number of `Object[]` to pass arguments. This is because `Compound Procedures` are a subclass of `ProcedureN`. One approach could have been to create classes `Lambda[01234N]` and `Compound[01234N]`. However, `Compound` needs to extend the `Environment`, which contains an `Object[]` of values. To address this, new `Environment[01234N]` classes could be introduced but then the `LexicalAddress Expression` would go from using an array reference to an overloaded method call to fetch its value. Even with this, an `Environment[01234N]` object is still allocated. Obviously another approach is needed.

5.1 `let` Optimization

One partial solution to this dilemma is to use `let` optimization. To understand how this works, recognize that:

```
(let ((a x) (b y) (c z)) ...)
```

is compiled as

```
((lambda (a b c) ...) x y z)
```

In the `let` case, there is no need to extend the `Environment` with a new frame. This is true of any case of a `Lambda Expression` compiled in a `Application Expression` operator position.

To understand why this is true, remember that when a `lambda` is evaluated, it creates a `Compound Procedure` that remembers the `Environment` that it was evaluated in so that when the `Compound Procedure` is later evaluated, perhaps in a different context, its original `Environment` will be used.

However, in this case it is not possible that the `Compound Procedure` would ever be used in another context, because it is in the operator position of an `Application Expression`.

To perform the `let` optimization, the expression:

```
((lambda (a b c) ...) x y z)
```

is converted to:

```
(begin (set! a x) (set! b y) (set! c z) ...)
```

Obviously the values of `a`, `b`, and `c` need to be stored somewhere, so the encapsulating `CompileTimeEnvironment` is expanded to contain space for these new variables. To support this, the `CompileTimeEnvironment` is changed from containing a `Symbol[]` to containing a `Variable[]`. `Variable` is a new class that tracks a `Symbol` name and in addition whether the `Variable` should be considered live or dead in the current lexical environment.

When compiling the body of the `let` optimized expression, the `Variables` that are newly extended in the current `CompileTimeEnvironment` are live, but are marked as dead after the body is compiled. The `CompileTimeEnvironment` also needs to be changed to search its list of variables from right to left, instead of the previous left to right. These changes cover the cases of the new bindings obscuring older ones with the same name, ensuring that the inner most one is found if it is still alive, or the outer ones being found if it is now dead.

Here is an example of how a dead variable can happen with `let` optimization. In the expression:

```
(let ((a 1))
  (+ (let ((a 2)
       a)
     a)))
```

the two Environment frames can be merged, based on the above discussion, into one that looks like this:

```
(let ((a1 1))
     ((a2 (unspecified)))
    (+ (begin
         (set! a2 2)
         a2)
        a1
```

In practice, the two Variable a's are not renamed a1 and a2. The expression a2 would be compiled in an CompileTimeEnvironment where the variable a2 would not be present. Then a2 would be added to the CompileTimeEnvironment as a live Variable. The body of the inner let, transformed into the a2 at the end of the begin, would then be compiled in an Environment where the a2 would be visible as a live Variable. After compiling the body of the inner let, the Variable a2 is marked as dead. Then when compiling the final reference to the Variable a, rewritten as a1, the original outer a is used, skipping the now dead inner Variable a.

In addition, the top level of the Compiler wraps the top level expression, e, with an empty environment, like this ((lambda () e)). This ensures that the Compiler will have an Environment to move let optimized bindings out into.

Before this optimization, mulitple Object[] and multiple Environments would be allocated, especially for a let* expression. Afterward, for the let* case for example, only one larger Object[] and single Environment would be created.

5.2 Closure Analysis

While the let optimization reduced the number of run-time allocations by removing many uses of Compound in rewritten syntax, it did not eliminate the allocations for ordinary function calls.

In the let optimization case, it was easy to see that the lambda did not require a Compound Procedure, also known as a closure, to be created. However, to do this in the general case, the Compiler needs to perform what is known as closure analysis.

This `Compiler`'s closure analysis involves deciding which variables can be stored on a simple stack as opposed to heap-allocated frames. Most languages use a simple stack for all allocations. However, Scheme functions may access variables outside the scope of their lambda definition in an encosing lexical environment. In the following example, the function bound to `counter` references the variable `n` in its local scope and the variable `count` in its enclosing lexical environment:

```
> (define counter
    (let ((count 0))
      (lambda (n) (set! count (+ count n)) count)))
> (counter 1)
1
> (counter 1)
2
> (counter 2)
4
> (counter 4)
8
>
```

Without closure analysis, the `lambda` would create a `Compound Procedure` in an `Environment` containing the `LexicalAddress count`, and when the `Compound Procedure` is applied, another new `Environment` is created containing the `LexicalAddress n`.

With closure analysis, the goal is to avoid allocating an `Environment` for the variable `n`, by passing the value on a more traditional stack, making `Compound Procedures` perform as well as primitive `Procedures` in the common case, which is when none of their variables reference external lexical environments.

5.2.1 `Stack`

In order to take advantage of the benefits of closure analysis, the implementation needs to avoid allocating storage when passing an arbitrary number of arguments on the stack. Unlike the C programming language, Java does not support n-ary arguments. Instead, apply will change to use a new `Stack` data-structure to cheaply pass arguments.

The `Stack` class is sort of a hybrid between a `Vector` and an `Object[]`. Like a `Vector`, it automatically handles resizing issues. However, like an `Object[]` it allows cheap access directly to its elements and its currently `inUse` length. It also provides storage for the current `frame` index.

`Procedure` application is changed once again, resulting in this new API:

```
abstract public Object apply0 (Stack s) throws ScriptException;
abstract public Object apply1 (Stack s) throws ScriptException;
abstract public Object apply2 (Stack s) throws ScriptException;
abstract public Object apply3 (Stack s) throws ScriptException;
abstract public Object apply4 (Stack s) throws ScriptException;
abstract public Object applyN (int n, Stack s) throws ScriptException;
```

For the `apply` special cases the number of arguments is encoded in which method is called. For the n-ary case, a separate count `n` is passed in to replace the length of the `Object[]`.

To show how this change affects the example used above, here again is an example of the `Cons` primitive `Procedure`:

```
public Object apply2 (Stack s) {
    Object o1 = stack.array[stack.inUse-1];
    Object o2 = stack.array[stack.inUse-2];
    return new Pair(o1, o2);}
```

For debugging, `Stack` implements `toString` to dump the stack and its contents using the `Writer`, which was invaluable in debugging the transition to argument passing on the `Stack`, as well as the subsequent closure-analysis work.

5.2.2 `Until`

Closure analysis involves analyzing any forms that can introduce new bindings. One other form of kernel syntax in the implementation that can introduce new bindings is the `Do Expression`.

To simplify the analysis, the `Do Expression` can be broken down into a traditional `let` and a new kernel `Until Expression`. For example this:

```
(do ((x y (+ x 1))
     (a b))
    ((foo? x) ...a)
  ...)
```

can be converted to:

```
(let ((x y)
      (a b))
  (until (foo? x)
         ...
         (set! -x (+ x 1))
         (set! x -x))
  ...a)
```

making it amenable to closure analysis.

The `Until Expression` simply evalutes its first sublist as a termination condition. If the value is true, the loop exits. If it is false, the rest of the sublist is evaluated. Then the termination condition is tested again as the loop repeats.

5.2.3 Closure Analysis at Compile Time

Finally, with argument passing switched over to the `Stack` and the removal of the `Do Expression`, the `Compiler` can be extended with closure analysis.

First, the `Variable` class used in the `CompileTimeEnvironment` is extended to indicate whether the `Variable` is to be heap or stack allocated.

Second, two new `Expressions`, `LocalAddress` and `LocalAssignment`, are added to represent getting and setting values on the local `Stack`, as opposed to the lexical `Environment`. In addition, `LocalAddress` and the existing `LexicalAddress` now remember a pointer to the `Variable` object, not just the `Symbol`.

Third, the algorithm for searching the `CompileTimeEnvironment` for a variable is changed. Previously, if a variable was found in the `CompileTimeEnvironment`, it meant it was a `LexicalAddress`, otherwise it was a `GlobalVariable`. Now a `CompileTimeEnvironment` might contain live or dead `Variables`, `Variables` that are known to `Environment` allocated, or those that are potentially to be `Stack` allocated.

As before, the search starts at the innermost `Environment` and moves out. Within an `Environment`, `Variables` marked as dead are skipped in the search. As before, `Variables` without a matching `Symbol` name are also skipped.

When a matching name is found, there is still more analysis to be done. If the search is no longer in the innermost `Environment` frame, this variable is now known to require `Environment` allocation. The `Variable` is changed to mark it as heap allocated. This would correspond to compiling the `Variable count` in the `set!` expression in the above example.

If the `Variable` is known to be heap allocated, a `LexicalAddress` is now created as before. Otherwise, a `LocalAddress` is assumed to be okay and is returned.

Fourth, a second pass is now added to the compiler which fixes up any incorrect assumptions of `Variables` as `LocalAddresses` that are later found to be `LexicalAddresses`. Note that in the above example, the first time that the `Variable count` is compiled in the `let`, the compiler would not yet realize that it needs to be a `LexicalAddress`, so it would make a `LocalAddress` on the first pass. Since the `LocalAddress` remembers its `Variable` object, it is simple to fix up the tree whereever a `LocalAddress` references a heap allocated `Variable`.

To implement this second pass, the `Expression` class is extended with a `fixupVariables` method:

```
abstract public Expression fixupVariables (
    CompileTimeEnvironment environment);
```

At the top level of the `Compiler`, `fixupVariables` is called and the new return `Expression` is returned. Most `Expressions` simply call `fixupVariables` on their sub-`Expressions`, replacing the old sub-`Expressions` with potentially new ones, and then simply return themselves. The main departure from this general rule is that `LocalAddresses` referring to dead `Variables` create and return new `LexicalAddresses` to replace themselves. `fixupVariables` also fixes up variable assignments, since if within `LocalAssignment` fixing up the `LocalAddresses` changes it to a `LexicalAddresses`, the `LocalAssignment` replaces itself with a `LexicalAssignment`.

Another function of `fixupVariables` is to calculate the frame and stack offsets for `LexicalAddresses` and `LocalAddress` respectively, which are not fully known in the first pass. Any attempt to calculate them in the first pass could fail because any change of a `Variable` from stack to heap allocation could invalidate offsets calculated earlier in the pass. To make this work in the second pass, `Lambda` now remembers its `CompileTimeEnvironment` so that when it calls `fixupVariables`, it can reinstate the proper `CompileTimeEnvironment` for fixing up its sub-`Expressions`. `Lambda` also calculates the number of stack allocated and heap-allocated variables, for run-time use by `Compound Procedure` in pushing space on the `Stack` and allocating `Environment` frames.

5.2.4 Closure Analysis at Run Time

At run time, closure analysis means updating the signature of `Expression.eval` once again. In addition, `Compound.applyN` needs to be updated to take advantage of the new information from closure analysis.

`Expression`

It seems that if `apply` changes `eval` has to change with it. `Expression.eval` is changed once again to now pass the run-time Stack.

```
public abstract Object eval (Environment e, Stack s);
```

Most `Expressions` simply pass this `Stack` unchanged when evaluating sub-`Expressions`. The obvious users are `LocalAddress` and `LocalAssignment`, that get and set values from the `Stack`, relative to the current `Stack frame` index. In addition, all the varieties of `Application.eval` now push the values from their evaluated operands onto the `Stack`. After calling their operand `Procedures`, they pop the pushed values off the stack by resetting the inUse index.

`Compound`

Significant changes were made to `Compound.applyN`. As before it validates the number of arguments passed matches what its `Lambda Expression` expects. It also remembers the previous `Stack frame` index and then moves the `frame` index to the current `inUse` end of the `Stack`. After that, the code becomes more complicated.

The next part of `Compound.applyN` deals with setting up the `Environment`. Remember that in the second pass the number of heap-allocated variables required was calculated and stored in the `Lambda Expression`. If no heap-allocated variables are required, `Compound` simply sets the current Environment to the one where the `Lambda` was defined without extending it. This is the best case that results from closure analysis. However, if heap allocation is required, a `Environment` is created to hold only the heap-allocated variables, which are copied from the stack into this new `Environment`, which is then used for evaluating sub-`Expressions`.

After the environment is setup, additional empty slots are pushed onto the stack for any local variables needed during the evaluation of the body of the `Lambda`, as a result of `let` optimization. With that, the body is evaluated with the calculated current `Environment` and `Stack`, after which the locals are popped off.

Finally, the `Stack frame` index is restored to the saved value and the result returned on the Java stack.

5.3 `Quoted`

When the `SelfEvaluating` and `Quoted Expressions` were removed from use as a wrapper for run-time values, `Expression.eval` needed to be changed to perform an `instanceof` check to distinguish between `Expressions` and simple quoted values. Overall this change was good, removing unnecessary run-time allocations. However, where quoted values really are needed, it traded off a single compile-time allocation for an `Expression` for a run-time `instanceof` check every time any `Expression` of any type is evaluated.

The Java profiling tools pointed out the cost of this. The `Quoted` was resurrected to be used whenever whenever quoted s-expressions are compiled, as well as any non-`Pairs` returned from the `Reader`, such as `Constants`, `Booleans`, `Integers`, `Doubles`, `Characters`, `Strings`, and `Vectors`.

5.4 Removing Implicit `Begin`

Another waste of processing time found by the profiler was the use of implicit `Begin Expressions` in the `Compiler`. For example, when compiling a `Lambda` or `Until`, the body was compiled by wrapping the body in a `(begin ...)` and invoking the `Begin SpecialFormCompiler`.

However, at run-time, this meant when `Compound` or `Until` was evaluated, there was an extra level of method call over head for `Compound.eval` and `Until.eval` to each invoke `Begin.eval`. Instead the simple loop from `Begin.eval` was inlined into these classes' `eval` methods.

In addition, this affected the `Expression.toString` implementation for these classes, because they would print out the implicit `Begin` when they called `Expression.toString` recursively on it. When the `Begin` `Expressions` were removed, a static method was added to `Begin` to convert `Expression[]`s into `Strings`.

5.5 Analysis of Third-Pass Implementation

The third pass has really started to provide a more mature environment. Several unmentioned small bugs were reported and fixed. Both space and time performance was analyzed and optimized. However, as always, there are still issues to explore.

5.5.1 Analysis of `let` Optimization

`let` optimization removes unnecessary extra frames. However, imagine the case:

```
(define foo
  (let ((a (cons-really-large-list-structure)))
    (let ((b (car a)))
      (set! a (cdr a))
      (cons (lambda ()
              (set! a (cdr a)))
            (lambda ()
              (set! b (cdr b)))))))
```

`foo` is a `pair` where the `car` and `cdr` are both functions. The function in the `car` of `foo` is closed over the variable `a`, while the function in the `cdr` of `foo` is closed over the variables `b`. At this point neither `a` nor `b` can be garbage collected, because they are captured by the `car` and `cdr` of `foo` respectively.

Evaluating the expression `(set-cdr! foo '())` would remove all references to the variable `b`, allowing it to be garbage collected. However, if instead, the expression `(set-car! foo '())` is evaluated, it would not remove all references to the variable `a`. This is because the `cdr` still references `b` in a frame that referencs the frame containing `a`.

`let` optimization would rewrite the above as:

```
(define foo
  (let* ((a (cons-really-large-list-structure))
         (b (car a)))
    (set! a (cdr a))
    (cons (lambda ()
            (set! a (cdr a)))
          (lambda ()
            (set! b (cdr b))))))
```

After `let` optimization, the `car` is now preventing the variable `b` from being garbage collected even if the `cdr` of `foo` is cleared. This is because one frame contains both the variables `a` and `b`, whereas before the `car` only held the variable `a`.

One solution might be to have the closure create a copy of only what it needs from the environment, not the entire environment, at run time. Or perhaps the compiler could arrange at compile time to have separate environments at run-time by using a new environment representation.

5.5.2 Analysis of Closure Analysis

Once again the lack of tail recursion became an issue. If tail recursive function calls worked, there would have been no need to support first `Do`, and now `Until` as kernel special forms.

The benefits of having special-cased `Application Expressions` are now less clear. The implementation now avoids allocating `Object[]s` for all primitives by using the new `Stack` class for passing arguments. However, it does prevent having to pass an argument count in most cases, so for now it remains.

A new `break-point` primitive was added when debugging closure analysis. By placing it in various complicated expressions, it allowed the `jdb` debugger to be stopped in precise places so that `Expressions` and the new `Stack` structure could be more easily studied at run time.

Chapter 6

Fourth-Pass Implementation

The third-pass implementation was deployed unchanged through several major revisions of its embedding application. In the fourth pass, some performance work was done, although that was not nearly the focus it was in the third pass. Instead, attention shifted to the new needs of the embedding application. One new requirement for the implementation was running in a Java `Applet` environment in a web browser. Another new requirement was allowing Scheme code to dynamically invoke Java via the new reflection API.

6.1 `Applet`

The biggest new requirement was for the implementation to work in the Java `Applet` environment, as opposed to simply in Java applications. The primary goal was allow for a GUI tool to be constructed that allowed a programmer to experiment with Scheme hooks and see their impact without running them on a production server.

6.1.1 `java.net.URL`

One of the main restrictions on `java.applet.Applets` is the inability to do file I/O. However, the implementation relies on loading `scm` files to initialize itself. Fortunately, an `Applet` is allowed to read from `java.net.URL` objects, with the restriction that the URLs are references back to the server from which the applet was downloaded.

The `Script` class is changed to contain a base URL, from which other relative URLs can be loaded. `Script.load` API was changed to take either a fully qualified URL or a relative URL `String` to be resolved with `Scripts's` base URL. The Scheme `load` function was similarly changed to expect URLs.

The Java application environment can then initialize Script's base URL with a file URL, while in the `Applet` environment, it can be initialized using `Applet.getCodeBase`.

In addition new Java extensions for manipulating URLs such as `as-url` were added to convert from absolute or relative URL `Strings` into URL objects. URLs are almost as important as `Date` in modern systems, so having these new extensions is generally useful.

6.1.2 Syntax Checking

One new feature to support the GUI tools was syntax checking. Basically this means compiling to an Expression tree without then immediately evaluting the Expression. This was easily added as a new `Script.compile` API. It also allows an `Expression` to be compiled once and remembered in a Java variable, and then repeatedly executed later. Before this, `Procedures` were multiply applied, but arbitrary `Expressions` could not be repeatedly evaled.

6.1.3 `ScriptException`

Until now, `Script.eval` was the only way to evaluate arbitrary Scheme code. However, this was very console-centric, expecting to report warnings and errors to a `PrintStream`. Now the implementation is running in a GUI environment so a cleaner way to report errors is needed.

First, a new API, `Script.evalWithException`, was added. This is the core logic from `Script.eval`, minus the console-centric code for handling `ScriptException`. The new API throws `ScriptException`, allowing the caller to choose how to display the error.

However, there still remains a problem with Scheme stack traces. As mentioned above, when there is a `ScriptException`, `Expression.toString` is called on each `Expression` as the stack is unwound, with the result displayed to the console.

To remedy this problem, `ScriptException` is extended with a `Vector` of `Expressions`. Instead of calling `toString` on an `Expression` as the stack unwinds, the `Expression` itself is just added to the `Vector`. A caller can then choose to examine this `Vector`, or use the new `ScriptException.stackTrace` method to convert the stored stack trace into a `String` for display. This `stackTrace` method also includes any Java stack-trace information for `PrimitiveExceptions` where something went wrong in the execution of Java code called from the Scheme code.

6.1.4 Script Widget

To pull all of this together, a special UI widget was designed for editing Scheme. This started a simple text widget with parenthesis matching. This was combined with a button hooked up to the new syntax checker. If there was a problem checking syntax, the `ScriptException` could now be asked for its stack trace, which could then be shown in a dialog box, instead of the hidden Java console. In addition, a simple pretty printer was added on another button to do simple automatic indenting of Scheme code.

6.2 Reflection

Until now new primitives were added as subclasses of `Procedure` because not many other alternatives were available. However this meant it was hard for end users to add access to their own Java code because Procedure was not made part of the public API. In hindsight this seems to have been a good choice, given how much `Procedure.apply` has changed through each implementation pass.

6.2.1 `java.lang.reflect`

Until now, the implementation worked with the Java 1.0 API. At this time, the embedding application moved to the Java 1.1 API. One of the additions to the 1.1 API was the `java.lang.reflect` package, also known as reflection.

Reflection allows a Java program to dynamically access fields and invoke methods of classes by `String` name without having a statically compiled knowledge of those fields or methods. What this means to the Scheme implementation, is that a user can define Scheme primitives by specifing by name the class and member they want to access.

6.2.2 Reflection Extensions

To bootstrap the reflection extensions for Scheme, only three simple primitives are required. The first, `class-for-name`, converts from a `String` class name to a `java.lang.Class` object. The second, `class-get-method`, looks up a method object using a `String` method name and a list of method argument `Classes`, to disambiguate overloaded methods, and returns a `java.lang.reflect.Method` object. The third, `method-invoke`, allows a `Method` object to be invoked with an object for instance methods or `null` for static methods, as well as a list of arguments to the method, returning an `Object` which is the `Method` call's result.

Now that `class-for-name` allows for the creation of `java.lang.Class` objects, the `new` and `instanceof?` extensions are changed to use these `Class` objects instead of simple class names. Here is an example of how `pair?` shown above, was redone:

```
(define Pair.class (class-for-name "Pair"))
(define (pair? x)  (instanceof? x Pair.class))
```

One problem with reflection is that looking up the Method with `class-get-method` each time `method-invoke` is called is expensive. To resolve this, the `Method` object is cached in a closure. The real API for people to callers to use is then defined as follows:

```
(define (make-method class name . parameterTypes)
  (let ((method (apply class-get-method class name parameterTypes)))
    (lambda x (apply method-invoke method x))))

(define (make-static-method class name . parameterTypes)
  (let ((method (apply class-get-method class name parameterTypes)))
    (lambda x (apply method-invoke method '() x))))
```

Although this shows the API for methods, it does not demonstrate access to fields, which is part of the reflection API. However, given these primitives, it is possible to reflect the reflection API itself to access the methods for looking up `java.lang.reflect.Field` objects from a `Class`:

```
(define class-get-field
  (make-method Class.class "getDeclaredField" String.class))
```

as well as to reflect the APIs for manipulating the result Fields objects:

```
(define field-get
  (make-method Field.class "get" Object.class))
(define field-set
  (make-method Field.class "set" Object.class Object.class))
```

Of course, looking up `Field` objects every time `field-get` or `field-set` is called is expensive, just as with `class-get-method` and `method-invoke`. So once again, the resulting `Field` object can be cached is a closure as well:

```
(define (make-field-getter class name)
  (let ((field (class-get-field class name)))
    (lambda (obj)
      (field-get field obj))))

(define (make-field-setter class name)
  (let ((field (class-get-field class name)))
    (lambda (obj value)
      (field-set field obj value))))
```

Defining the full reflection API using a subset of the reflection API hopefully demonstrates the power of reflection. In addition to what was shown, there are parallel APIs to methods for constructors: `class-get-constructor`, `make-constructor`, `constructor-new`.

As a final example, when an API for array manipulation was needed, it was easy to add entirely in Scheme:

```
(define array-new
  (make-static-method Array.class
                      "newInstance"
                      Class.class
                      Integer.TYPE))
(define array-get-length
  (make-static-method Array.class
"getLength"
```

```
                          Object.class))
(define array-get
  (make-static-method Array.class
                      "get"
                      Object.class
                      Integer.TYPE))
(define array-set
  (make-static-method Array.class
                      "set"
                      Object.class
                      Integer.TYPE
                      Object.class))
(define (list->array lst class)
  (let ((c (length lst)))
    (let ((a (array-new class c)))
      (do ((i 0 (+ i 1))
           (l lst (cdr l)))
          ((= i c) a)
        (array-set a i (car l))))))
```

6.2.3 Reflection Performance

Given the availability and power of reflection, it seems like the implementation might be able to reduce the number of primitive `Procedures` written in Java to `new`, `class-for-name`, `class-get-method`, and method-invoke. However, this was not done because of the overhead of using reflection versus using Java code directly.

The example below defines `reflective-car` as a version of `car` that uses reflection. Timings are performed using a ten million iteration `do` loop with an inherent overhead of 12 seconds. If the body of the loop simply accesses a quoted constant, the time goes up to 13 seconds. If the body uses the traditional `car` the time increases to 15 seconds. However if the `reflective-car` function is used, the time increases ten-fold to 155 seconds.

```
> (define reflective-car (make-field-getter Pair.class ‘‘car’’))
```

```
> (define pair (cons 1 2))
> (time (do ((i 0 (+ i 1))) ((= i 10000000))))
12
> (time (do ((i 0 (+ i 1))) ((= i 10000000)) '()))
13
> (time (do ((i 0 (+ i 1))) ((= i 10000000)) (car pair)))
15
> (time (do ((i 0 (+ i 1))) ((= i 10000000)) (reflective-car pair)))
155
>
```

Even if performance was not a concern, additional Java primitives would be necessary besides those listed above. The reason for this is that almost all Java operators such as `+` are not available through method calls. The main exception is `instanceof` operator for which the functionality was exposed as `Class.isAssignableFrom` in JDK 1.1.

6.3 Multi-engine

Until now, there was a limitation of one scripting environment per Java virtual machine. This was largely because of the accumulation of global state such as the first `GlobalEnvironment`, the more recent list of `RuntimeExceptions` to rethrow, and the new base URL.

However, there was a new application requirement to have multiple isolated `Script` engines simultaneously. In order to accomplish that goal, all global state needed to be removed.

The strategy was to make the `Script` class the new repository for previously global state. Each `Script` engine would be represented with an instance of the `Script` class. `Constants` such as `Null`, `EOFObject`, `Unspecified`, etc., could still be shared across the engines. The static fields for the `GlobalEnvironment` and base URL were changed to instance URLs. Then these changes needed to be propagated further.

`Compiler` had previously referenced the `GlobalEnvironment` to register its `SpecialFormCompilers` and for defining new `GlobalVariables` for `Variables` not found in its `CompileTimeEnvironments`. `Compiler` itself moved from being static to being an instance. An instance was created and referenced from the `Script` instance. The `Compiler` instance maintains a back pointer to its `Script` instance. Most of the static methods of `Compiler` were changed to instance methods so that they could access the `Script` instance.

The `Loader` class already was used through instances, because each `Loader` already had its own `Reader` instance. The `Loader` did however statically access the `Compiler`, so now the `Loader` was modified to remember a `Compiler` instance to use.

Like `Compiler`, many of the `Script` methods making up the Java-to-Scheme API changed from static to instance methods so the caller would be forced to specify which `Script` engine to use. This was required because API methods such as `Script.eval`, `Script.evalWithException`, `Script.compile`, `Script.load`, `Script.lookup`, and `Script.call` were simple wrappers around the `GlobalEnvironment`, `Compiler`, and `Loader`. The notable exceptions to this conversion from static to instance were the numerous type marshalling methods such as `Script.object`, `Script.string`, `Script.pair`, etc., which remained unchanged.

6.3.1 Procedures

There were some issues in pushing the change through some of the primitive `Procedures`. For example, `as-url` needs access to the `Script` base URL to produce URLs relative to the current `Script` engine. `define-rewriter` needs access to the `GlobalEnvironment` to define new macros. `eval` needs access to the `Compiler` to translate s-expressions into `Expressions`. `load` needs access to the `Script` itself to call `Script.load`. The primitive `Procedures` need a way to access this `Script` state from their `apply` arguments.

The solution to this issue is to add a `Script` instance field to the `Stack` class. This allows all primitives to access the global `Script` state though their existing `Stack` argument, which means not having to change the signature of `Procedure.apply` yet again. Also it is conceptually clean, since the `Stack` represents the current state of execution, which naturally includes which `Script` engine created this `Stack`. `Environment`, the second choice, was not as good because `Environment` really is a nested set of `Environment` frames, so an extra reference of memory would be added to each frame, instead of just the single `Stack` instance. With this change, all references to global state were removed from the implementation, allowing multiple scripting engines to peacefully coexist in one Java virtual machine.

6.3.2 Thread-Local Storage versus `Stack`

Now that each thread has its own instance of a `Stack`, the thread-local storage implementation was replaced with new `Stack` instance fields. Although not as extensible, this means the cost to access the I/O state is reduced to a field reference from a `Thread.currentThread` lookup as well as two `Hashtable` lookups. The old thread-local storage implementation was kept for application use. If desired, the new JDK 1.2 `java.lang.ThreadLocal` implementation could be accessed via reflection, but since browsers only support JDK 1.1, and only partially at that, depending on this new API was avoided.

6.4 Internationalization

Another new requirement of the embedding application in this pass was to support internationalization. Partially this means adding primitives for new Java 1.1 classes such as `java.text.MessageFormat`, but existing primitives need to be updated to be aware of internationalization issues as well.

One of the major updates was to use the new character-oriented `Reader` and `Writer` classes in place of the older byte-oriented `InputStream` and `OutputStream`. This meant changing from `PushbackInputStream` to `PushbackReader` and from `PrintStream` to `PrintWriter`. It also meant changing the transcript support from `MultiPrintStream` to a new `MultiPrintWriter`.

As mentioned, the difference between the APIs are method signatures using characters instead of the more traditional bytes. In fact the Scheme standard already uses the general term character, avoiding the term byte altogether. Moving to the new internationalized APIs that can deal with any Unicode characters is definitely in the spirit of the Scheme standard.

However, because the Scheme standard does not mention bytes, it does not specify how to map characters into bytes, leaving that decision up to the implementation. The Java API includes `String` encoding arguments to define various standard algorithms for converting characters to and from bytes. A Java virtual machine has a default encoding to use when none is provided, and `open-input-file` and `open-output-file` are changed to use this default. In addition, `open-input-file` and `open-output-file` are extended to take an optional argument to allow Scheme programmers to specify the Java encoding of their choice. If a Scheme programmer needs to manipulate files of bytes, they can use a

8-bit single byte encoding such as ISO-8859-1.

6.5 Performance

As always, there is more performance work to be done. Fortunately the issues become smaller and smaller, more tweaking than structural changes.

6.5.1 `GrowOnlyHashtable`

As mentioned before, synchronization can be a bottleneck. The standard `java.util.Hashtable` includes synchronization by default. Even a copy of this class stripped of synchronization need to be synchronized if instances may be shared across threads such as for the `GlobalEnvironment` or the `Symbol` table.

A new data-structure called `GrowOnlyHashtable` is used to avoid unnecessary synchronization. The `GrowOnlyHashtable` is specially constructed to not require synchronization on the `get` method. No synchronization is required on the `put` method if it does not matter which object ends up in the `GrowOnlyHashtable`. However, since the `GlobalEnvironment` and `Symbol` table need to have unique values in the `GrowOnlyHashtable`, special synchronization is required around the `put` method in these cases. However, overall since most access is through `put` and not `get`, this cuts down significantly on the number of synchronizations.

6.5.2 `new Integer`

As mentioned before, the implementation uses `java.lang.Integer` to represent Scheme `integer` values. However, Java mathematical operators work on `ints`, not `Integers`. The Scheme math primitives use `Integer.intValue` to convert to `ints` to perform the operation. Until now, the implementation would convert from the `int` back to `Integer` by using the `Integer` constructor.

However, many of the `Integers` created are conceptually the same value. For example, many standard functions performing iteration keep small integer counts. Also, the `Reader` creates `Integers`, including many small constants such as 0 and 1 commonly used for iteration and incrementing.

It is safe to reuse `Integer` objects since they are immutable. For example, the same `Integer` can be used to represent zero in all cases because once an `Integer` is created, its `intValue` cannot be changed.

`Script.getInteger` is added to implement this reuse. Underneath, a range of small positive and negative integers is lazily allocated and cached in an `Integer[]`. No synchronization is required, because if two threads store two different `Integers` in the array element simultaneously, they will have the same `intValue`, and look equivalent externally. They look the same externally since pointer equality can still not be used to compare `Integers`, since `Integers` outside the cached range will be created each time they are needed.

This change not only increases integer math performance by removing allocation but has the side effect of speeding up numerous library functions that perform iteration.

This pooling of small `Integers` is an example of the Flyweight design pattern. [15]

6.5.3 `char[]` to `String`

As mentioned above, the type marshalling methods in `Script` convert `char[]` to `Strings` when passing objects into Java. Analysis discovered that 95% of these objects were repeated frequently, so a cache was added to avoid the unnecessary allocation. It is safe to reuse the `String` values because, like `Integers`, the values are immutable. A `GrowOnlyHashtable` was used, this time without synchronization on the `put` method, because if two of the same `String` are allocated the lack of pointer equality is not an issue, like `Integers` and unlike `Symbols`.

6.6 Analysis of Fourth-Pass Implementation

The fourth pass contained several incremental changes leading up to the present time. Besides discussing the impact of changes made in this pass, this section will summarize the remaining issues after the final pass.

6.6.1 `Applet` versus Reflection

In this pass both `Applet` and reflection support were added. However, another Applet restriction is on the Java reflection API. In a Java application, as opposed to an `Applet`, reflection can be used to access even `private` members of classes. This allows implementation of serialization and persistence APIs, but presents security problems in `Applet`.

In order to deal with this, as part of bootstrapping, the system tries to use the full reflection API using a simple `catch` extension. If it catches a `SecurityException`, it knows that it is running in the `Applet`, sets a global variable to indicate this, then switches to using the reduced public reflection API.

The `catch` extension looks like this:

```
(catch thunk class-name-string proc)
```

First, the `thunk` is run. If an exception is thrown that is a subclass of the class named in `class-name-string`, then `proc` is called with one argument, which is the exception that was caught. A `throw` function was added to match `catch`, which takes one argument, a subclass of `java.lang.Throwable`, to `throw`. There was no need to add a `finally` extension, since that is already provided by `dynamic-wind`.

There was one problem with this scheme for detecting `SecurityExceptions`. Microsoft Internet Explorer decides to throw a proprietary `com.ms.security.SecurityExceptionEx` instead of a plain `java.lang.SecurityException`. This is simple enough to work around, and was one of the few minor issues encounted with the Microsoft Virtual Machine for Java.

6.6.2 Primitives in `Applet` Environment

Even with a basic interpreter working in the `Applet` Environment, many primitives had problems calling restricted APIs. As mentioned, several of the I/O primitives that used `Files` had issues. Some other examples were the process and mail primitives which were forbidden from use in the `Applet's` sandbox.

The embedding application had to rework many of its primitives to use remote procedure call so they could run in the `Applet` environment. Fortunately, the application can use the `Applet` flag set during the bootstrapping of reflection to detect when this is required. For some primitives, such as for access to type 2 JDBC drivers, the functions would not even be defined when the implementation was in the `Applet` environment, because there was no hope those functions would work there.

In general moving to `java.net.URLs` from `java.io.Files` cleaned up a lot of issues that had plagued the old implementation. For example the differences between `File.separator` characters between Unix and Win32 required all `File` routines to canonicalize their `File.separators` so that scripts would work portably. With URLs, the details of file separators and other issues are hidden below the Java APIs.

URLs is a better API for describing files than simple Strings. It is much more reminiscent of the Common Lisp file-system neutral API. It allows programs to work across several different file sources without having to customize the application to understand each. [57]

Using URLs, which are absolute, removes the concept of current working directory which is problematic for two reasons. First, the current working directory is usually a process-wide concept, which complicates life for multi-threaded applications which might change the current working directory without anticipating the impact on other threads. Second, having code manipulate a global current working directory does not lead to nicely compartmentalized modules, since a program passing around relative paths cannot safely do so if the module might change the current working directory.

6.6.3 Multi-Engine versus `REPLServer` versus HTML

When the `REPLServer` was first created only one `Script` engine was allowed per process. Now that there could be multiple `Script` engines per process, it is not clear what the new semantics should be. One option would be to have each `Script` engine listen on a different port, meaning more configuration for the application. A second option would be to have the `REPLServer` be aware of all the `Script` engine and provide the user a choice when they connect, or a default `Script` engine and functions for switching between them.

In the case of the embedding application, an entirely new approach was taken. Instead of using a telnet-based UI, access to the `Script` engines was added to an existing HTML administration form. A text input of Scheme is posted for evaluation in an `javax.servlet.http.HttpServlet`, and the results presented back via HTML. Optionally, a file-upload input form can also be used to send a file to the server for evaluating.

The new `Script.evalWithException` and `ScriptException.stackTrace` added for the GUI were also valuable in constructing the new `HttpServlet` interface. Before, the `REPLServer` took advantage of its redirected I/O to send warnings and errors to the telnet client, which for the `HttpServlet` would have left the output on a potentially different machine. The `HttpServlet` uses the `Script.evalWithException` to evaluate the Scheme, and can render any warnings or errors including stack traces in the HTML result page.

6.6.4 Remaining Limitations to Scheme for Java

There still are several limitations of the implementation in its current state.

`Symbol`

`Symbols` are currently case-sensitive. This means that valid Scheme programs may not work if they reference standard functions using any uppercase letters or are not internally consistent in their symbol naming.

This implementation is not the only one with such a restriction. The Scheme Shell also uses case-sensitive symbols because it wants to map s-expression symbols into case-sensitive program-command arguments. [53] [54] This implementation chose to be case-sensitive for similar reasons, allowing for special reflection syntax to map from s-expression symbols to case-sensitive Java identifiers. Although this was not implemented, it is possible as a user `define-rewriter` macro.

As mentioned before, uninterned symbols are not supported. Often implementations have the non-standard `gensym` return uninterned symbols, but this implementation's `gensym` returns interned symbols. This could lead to namespace collisions for generated symbol names but has not been a problem so far.

`Reader` `vector` Syntax

The `Reader` does not support the little-known `vector` syntax of `#(1 2 3 4 5)`. This was a simple oversight that should be easy to correct.

Internal `define`

In Scheme, `define` expressions may appear at the beginning of the body of `lambda` and `let` expressions. These internal `defines` are syntactic sugar for `letrec`. The implementation has never supported these. Where it might have been used `letrec` was always used explicitly.

Tail Recursion

Perhaps the biggest limitation to traditional Scheme programmers is the lack of tail recursion as well as the related `let` loop. However, Java programmers writing extensions do not find this to be lacking. They detest the do loop, preferring instead to use a simpler while macro built with until, which is similar to the Java style of programming. This is clearly an important area for future work. A simple replacement for the `let` syntax rewritter could perform some Pseudoscheme style analysis to support the common case of named `let` loops.[47]

Limited Numerics

Scheme specifies a full tower of numerical types from `number` to `complex` to `real` to `rational` to `integer`. A conforming Scheme implementation is not required to implement the full tower, so strictly the fact that this implementation only provides `integer` and `real` support is not a violation of the standard.

Separate from the tower of numerical types, Scheme defines the concept of exact and inexactness. This implementation properly follows the rules for exactness so far as primitives that operate on exact values, in this implementation only `integers`, produce exact results. Specifically, the mathematical operations on only `Integers` produce `Integer` results while operations that mix `Integers` and `Doubles` produce `Double` results.

Scheme also encourages but does not require exact numbers of unlimited size. Since the implementation does uses the 32-bit signed `int` value inside a `java.lang.Integer` to represent its exact values, the size is currently limited. In JDK 1.1 Java introduced `java.math.BigInteger` as a new type of `java.lang.Number` so a Scheme program could replace the standard mathematical operators with ones that could handle exact `integers` of unlimited size as well. This was not provided because the application had no need for this feature. A similar approach could be used to incorporate `complex` and `rational` numbers as well.

`call-with-current-continuation`

A simple `call-with-current-continuation` implementation was added in this pass to provide for escape procedures. Internal to the `CallCC Procedure` which implements `call-with-current-continua` an `ExitProcedure` is created and passed to the caller's function. If the `ExitProcedure` is applied, the `ExitProcedure` stores itself and its argument in a special subclass of `ScriptException` called `CallCCException` which it then throws.

This thrown `CallCCException` is caught by the `CallCC Procedure` which then needs to consider two cases. If this `CallCCException's ExitProcedure` was the one created this `CallCC Procedure`, then the `CallCCException's` value is returned. Otherwise the `CallCCException` is rethrown to another `CallCC Procedure` waiting higher up on the stack.

Whenever an `ExitProcedure` is called or the program flow returns past the `CallCC Procedure` that created it, the `ExitProcedure` is marked as used to prevent its use for anything other than an escape procedure. If it is called after it is marked as used it returns `Script.Unspecified`.

Similar to tail recursion, this restricted implementation seems to disappoint traditional Scheme progammers more than Java progammers. Java programmers prefer to use the Java throw and catch extensions rather than the limited `call-with-current-continuation` implementation. Even with its limitations, the current `call-with-current-continuation` does satisfy most daily uses for Scheme programmers. In this implementation, the restrictions on `call-with-current-continuation` seem similar to those in Pseudoscheme which builds its implementation using Common Lisp block.[47]

Chapter 7

Java and Scheme

This section will take a high level view of Java and Scheme, based on the experience of implementing this system.

7.1 Java Advantages

Since the implementation language here was Java, the first section talks about its strengths.

7.1.1 Portability

One of the biggest claims made by Java is "Write Once, Run Anywhere". How does this claim hold up in real world use?

Development Environments

In the early days of this implementation at the end of 1996 and begining of 1997 there certainly were problems. First, there were compiler ambiguities. Code that compiled with Sun's JDK and Symantec's Visual Cafe did not compile with Microsoft's Visual J++. Surprisingly, this was often because J++ was a more strictly correct compiler than even Sun's `javac`.

The biggest problem in these early days was on the Macintosh, where Metrowerks CodeWarrior originally limited the length of package and classnames due to the Macintosh filename limit of 32 characters.

Although most of these issues were hammered out in the various Java 1.0 systems, Java 1.1 brought new issues. Grafting inner classes and other additions to the original `javac` compiler led to numerous bugs, which were visible not only in `javac` compiler, but the derivative compilers such as Symantec's `sj` compiler used by VisualCafe. In a recent version of Java 2 known as JDK 1.3, the orignal `javac` was thrown out and replaced with a research compiler from Australia fixing most of the compiler issues, including fixing several more ambiguities that were tightened up in the Java Language Specification.

`Applet` Environments

Numerous small JIT bugs hounded Netscape and Internet Explorer alike. Netscape's Java virtual machine did not provide a working Thread.join method or support casting from an `Object[]` to subclasses such as `String[]`. Once again, surprisingly, Microsoft seemed to provide a more faithful Java system.

The Macintosh was the worst of all possible worlds. Even when class names were shortened, the Metrowerks Java virtual machine could not support large `Applets`. Even if development was done on Win32 or Unix, serious `Applets` would hang Java virtual machines from Netscape Navigator, Microsoft Internet Explorer, as well as the offical reference implementation from Sun.

Server Environments

For early server side work, only Win32 and Solaris were even considered. Solaris required kernel patches to support the use of green threads over native threads. Eventually HP's Java virtual machine was stable enough to support server multi-threaded server applications as well.

Today, IBM's virtual machines are considered some of the best on any platform. Their recent virtual machines for Win32 offer the best server performance. They also support Linux on platforms from the x86 to the S/390. They also support their own operating systems such as AIX, AS/400, MVS, and VM. The biggest problems holding back IBM's virtual machine are small JIT problems that should be overcome with time.

Reality

So really, a more realistic claim would be "Write Once, Debug Everywhere".

7.1.2 Language

Beyond the hype surrounding portability, Java also claims to be superior because of its language design. Many people debate about the more traditional object-oriented issues regarding multiple inheritance or interfaces versus inheritance. This section will talk about the other issues that often get left on the way side.

Exceptions

Java's Exception mechanism is one of its biggest contributions to developing modular applications. This is saying a lot, since exception systems have in fact been around for years, including in Java's closest relative, C++.

Exceptions are important because they separate error detection from error handling. In anything but the smallest programs, these two concepts are likely to be distinct.

Take for example the evaluation API for Scheme in Java above. At first, the API tried to handle all problems internally, logging the problem itself to the Java console, returning Java `null`, as opposed to `Script.Null`, to report that an error had occured.

However, as the needs of the application grew, placing the error handling into the code doing the error detection was clearly wrong. It prevented the application from choosing the approriate handling for the error depending on the context, which grew to include a traditional graphical user interface and an HTML user interface, as well as a more command line oriented user interface.

So what makes Java's exceptions any different than C++'s exceptions? They both use `try` and `catch`, although Java adds the additional `finally` blocks, which are arguably sugar but nonetheless useful. They both allow the catcher to use inheritance to select related exceptions, instead of having to enumerate each specific exception. C++ manages, of course, to complicate things by differentiating between catching and throwing by pointer, by value, and by reference. C++ also extends things a bit, allowing not just classes but arbitrary types to be thrown, including things like `int` and `void*`, although this seems more confusing that useful.

One problem with C++ exceptions is that they were an add-on. Many compilers from `gcc` to Microsoft's `cl` have had trouble with them. Since the standard libraries predate exceptions, they do not use exceptions. These two problems combine to mean that C++ programmers do not tend to use exceptions. Without widespread use, exceptions do not achieve the potential of improving clarity and robustness of C++ programs.

One subtle advantage to Java exceptions is compiler checking. Although C++ allows a method to declare the exceptions that are thrown, it is nothing more than informational. For Java `java.lang.Exceptions`, which excludes `java.lang.Errors` and excluding `java.lang.RuntimeExceptions`, a method throwing an exception without internally catching it must declare it in its `throws` clause. This makes clear to the caller that a method that they called can throw an exception, since the caller must also choose to catch the exception or list it in its signature's `throws` clause.

Because the method writer must consciously choose to either handle or pass on an error, it is more likely that at some level exceptional conditions will be handled in at least a somewhat reasonable way, instead of the traditional way of C where a program that fails to check for an error code blindly continues on, usually resulting in an error downstream from where things really went wrong.

A Java class can certainly avoid this throws declaration by using an `Error` or a `RuntimeException`, and sometimes that is appropriate. `Errors` are used when application should not be expected to recover. `RuntimeExceptions` can be used if a widely used method needs to report a possible exception, but making virtually all methods declare that exception is seen as overkill, especially when it is known that a higher level framework handles the exception. But these cases are rare compared to the commonplace use of declared `Exceptions` as part of defining an API.

Garbage Collection

Garbage collection is part of the hype surrounding Java. Garbage collection is not a new concept, certainly not to Scheme programmers. However, it is worth mentioning the relationship between garbage collection and using a functional programming style.

Imagine a class like Java's `Number` with an `add` method, perhaps as an extension to support complex numbers. Supposed some code wanted to simply add a few numbers in a simple functional style like this:

```
Number e = new Number(x).add(new Number(y)).add(new Number(z);
```

This could be thought of short hand for:

```
Number a = new Number(x);
Number b = new Number(y);
Number c = a.add(b);
Number d = new Number(z);
Number e = c.add(d);
```

In C++, to cope with the manual deallocation, it is the even more verbose:

```
Number* a = new Number(x);
Number* b = new Number(y);
Number* c = a.add(b);
delete a;
delete b;
Number* d = new Number(z);
Number* e = c.add(d);
delete c;
delete d;
```

Note that in Java the simple version is correct, although in C++ the more verbose intermediate version is required so that pointers to intermediate values can be saved for later cleanup.

If things are this bad for functional composition of a simple `Number` class, they only get worse when combining several third party APIs, especially when error checking is added in for C++ libraries that are not using exceptions.

Packages

Java's package system is not sophisticated, but is better than nothing. It is based on declaring classes in nested packages, which most development environments map into nested directories containing Java source and class files. Organizations are encouraged to use their unique internet domain name as the outermost package to prevent namespace collisions. A little arbitrary perhaps, but it gets the job done, reducing naming conflicts to be within an organization, letting third parties work together without coordination. It encourages grouping of related classes into packages together, perhaps encouraging more structured system design, where systems with no namespace might dump all the classes into one directory, or at least a few shallow directories based on how the linker will assemble them into libraries.

One part of Java packages that leave something to be desired are the protection boundaries between packages. Here Java depends too much on its C++ heritage for guidance, with its `public`, `protected`, and `private` keywords, as well as its own the mysterious default protection provided when no keyword is used. While the `protected` and default permission allow any access from other classes in the same package, there is no way to grant permission to other packages without opening things up completely with `public`.

A single class can grant permission to its subclasses in another package to allow access, but other classes in the subclass's package have not ability to see the internals of this new class.

Arguably this is probably a good default to promote encapsulation. However, two packages cannot choose to cooperate privately together even if they want to. Supposed a package `com.foo.bar` provides a public API from company Foo to manipulate their `bar` interface. Suppose another package `com.foo.baz` wants to have full access to `private` member data in order to persist `bar` objects to a database. There is no way for `com.foo.baz` to grant a C++ like friend status to the package `com.foo.baz` or specific classes within.

Some approaches might be to have `com.foo.baz` subclass each of the classes from `com.foo.bar`, but then that would open up other outsiders to be able to do so. An application could replace the SecurityManager and use reflection to access the `private` members of `com.foo.bar`, but this is expensive, and removes any possibility of compile-time checking.

Immutable `Strings`

For all its oddities, Java's immutable `Strings` work out well for a couple of reasons. The first is that it is safe to pass them to library foreign code without worrying about the contents being modified. This also makes it clear that an API must make its own copy if it needs to side-effect the value which is often ambiguous without immutability. ANSI C and C++ provide the `const` keyword to specify that arguments are not to be modified, which Java does not provide, but that is sort of backwards, because it means the definer of the interface makes the promise, not the owner of the data, which seems to go against the object-oriented principles of data encapsulation.

One might wonder why `Strings` are special, since Java is not providing this form of protection to other common data-structures such as `Vectors` and `Hashtables`. One reason is that `Strings` are commonly used as keys in Hashtables, so guaranteeing that they are not corrupted is important for safety. A `java.lang.ClassLoader` might have a `Hashtable` mapping `String` class names to `java.lang.Class` objects. Imagine the havoc a program could cause that modified the `String` returned from `Class.getName`.

Another good effect of immutable `Strings` is that code is more likely to share `String` instances instead of making copies to prevent third parties from possible side-effecting values. This makes equality testing cheaper, since in many cases, the same `String` value will be represented by the same string reference, making the comparison as cheap as comparing numeric types.

One optimization that the sharing of `String` instances allows is when copies of objects are made. This type of shallow copying of objects while sharing immutable members might be common in an automatic persistance system such as an Enterprise Java Beans (EJB) container managing persistance of entity beans. The persistance system might keep one copy in a cache and make shallow copies for each transactional context. When a transaction is committed, the container will want to generate the minimal SQL to update only the fields that changed of the object. Not only is the initial shallow copy cheap, but the container can simply compare `Strings` using pointer equality instead of a more expensive `String.equals` operation.

Finally, one other optimization this allows is that `String.substring` can return new `String` instances that share an underlying `char[]`. The new `String` instance just has a different offset and length to indicate the part of the `char[]` it represents. While new wrapper `String` objects are created, the potentially larger `char[]` is shared.

7.1.3 Platform

Finally one more positive claim is the benefits of Java as a platform, not just a language. Sun has perhaps taken criticism for taking things too far at times, but having things like standard profiling and debugging APIs makes C++ compilers with incompatible name mangling algorithms look prehistoric.

7.2 Java Disadvantages

While Java seems to be a major step forward over C++, no language is perfect. Java has its shortcomings and pitfalls to beware of.

7.2.1 Threads

Threads are actually a good thing about Java. It is the first major language that has had threads as part of the language since its inception. What is bad about threads is their interaction with the standard I/O classes.

The major problem is supporting many simultaneous streams, such as in a server. The example in this implementation system is the `REPLServer`. The `REPLServer` has one `Thread` calling `ServerSocket.accept` looking for new connections. Whenever it has one, it quickly creates a new `Thread` to read requests from that new client.

The problem is this architecture of spawning a new `Thread` for each connection. It is fine for the `REPLServer` which is at most used by a couple of users at a time for debugging. However, imagine a chat system with thousands of clients concurrent connections. Because of possible firewalls between clients and the server, the clients need to remain connected to the server so they can receive their incoming messages. [45]

Many Java virtual machines have only simulated threads, so perhaps this architecture would be no worse for them than something more sophisticated. However, for Java virtual machines that map their `Threads` into native operating system threads, this turns out to be very expensive. Unfortunately most common server operating systems from commerical Unixes to Microsoft Windows NT cannot scale a single process to such large numbers of threads.

One solution is to provide an interface like BSD `select` or System V `poll`. [20] This allows a single thread to monitor several `InputStreams` simultaneously. Concurrency is still possible because it can feed a queue of ready `InputStreams` to a pool of `Threads` waiting to handle incoming requests. This pool of `Threads` can be used to throttle the concurrency in the server to make sure that the operating system is not swamped with excessive thread context switching.

There are more advanced APIs available today than `select` and `poll`. Microsoft Windows NT's I/O completion ports or Sun's `/dev/poll` can perhaps improve scalability even more, but they are even less portable. [64] There currently is a Java Specification Request for a new I/O API that could encapsulate all of these different platform specific interfaces. [27]

Some newer virtual machines try to use a virtual threads concept, where a number of process threads are mapped onto a potentially smaller number of operating system threads which are mapped onto a potentially smaller number of physical processors. This approach is taken by Solaris's Light Weight Processes.[59] While this seems to improve scalability somewhat, it is still not comparable to a less `Thread` intensive approach. [37]

Although it is desirable to expose this functionality in the most general manner possible, it can be hidden inside of an application server. Load balancing of I/O and queuing of work requests is not a new concept but a traditional part of transactional processing systems. [18] Weblogic uses native I/O code to improve performance by a factor of three in some cases. [5] If a more general solution is made available, application servers can avoid using their own native code to achieve scalability, leading to easier portability.

7.2.2 Synchronization

With threads comes the need for synchronization. Several of Java's classes such as `java.lang.StringBuffer`, `java.util.Vector`, and `java.util.Hashtable` include built-in synchronization that guarantees these data-structures cannot be corrupted by side-effects from multi-threads. Since important classes such as `ClassLoaders` might use these data-structure classes, a secure library is part of the requirement for the `Applet` sandbox.

So what could be wrong with that? The problem is that synchronization is not for free. What is good for security in an `Applet` starts to be a burdensome cost in a multi-threaded server application.

Implementation Problems

In the Java programming model, any `java.lang.Object` can be used for synchronization. A simple implementation might store a lock in each `Object`, however, this means an extra word of storage in each object which seems like an unacceptable tradeoff. So instead the

Sun reference virtual machines contain an internal hashtable from object handles to locks for those objects. While this cut down on the per object memory cost, it means that any synchronization, even from `Threads` synchronizing on unrelated objects, were bottlenecked by unknowingly synchronizing on this internal hashtable.

Some newer virtual machines such as IBM's get rid of this by simply adding the dreaded word of memory to each object. This is not as bad as it seems, because in IBM's new object layout, they also removed the use of handles to objects when they moved to a new garbage collector, so the amount of memory used ends up the same. [58]

Another more middle-of-the-road approch for virtual machines that use handles is to use some bits in the handle to index first into several tables instead of one. Although this does not eliminate contention, it does statistically lower the chances that two `Threads` might clash for unrelated objects.

`StringBuffer`

In the original Java Language Specification, the use of the `+` operator on `Strings` was defined as sugar over use of `StringBuffer`. This means that methods full of the `String` `+` sugar are synchronizing even though none of the values involved in the expression could possible be available to other threads. This is ridiculous since this is probably the most common use of `StringBuffer`. [16]

Of course there are cases when applications use `StringBuffer` outside of `String +`. Several third-party libraries provide their own implementation without the synchronization, such as Netscape IFC's `netscape.util.FastStringBuffer`.

Apparently Sun has partially seen the error of their ways. Newer versions of the Java Language Specification are have changed their wording regarding the `String +` operator, implying that the behavior should be like using `StringBuffer`, but not necessarily requiring its specific use. The new section on `String +` optimization is clear to point out that the compiler is free to use its own implementation in place of `StringBuffer`, and even to use its own implementation of routines for converting primitive types to characters without using `String.valueOf` methods that require an extra intermediate `String` operation. [17]

Unfortunately, it would have been more useful for Java to have included its own new non-synchronized `StringBuffer` variant that supported these `char[]` based formatters. Instead applications that want to be efficient in their `String` formatting are required to provide their own implementations derived from the `String.valueOf` implementations. Even worse, is that since no new standard class is involved, that means that compilers wishing to avoid `StringBuffer` have to inline code to do the optimization, potentially leading to code bloat.

In the final analysis, `StringBuffer` synchronization does not make much sense at all. Unlike `Vectors` and `Hashtables` which are often used as data-structures shared between `Threads`, no common application of a shared `StringBuffer` comes to mind. Perhaps this is once again simply taking the `Applet` sandbox safety too far.

`Hashtable` and `Vector`

Built-in synchronization is more valuable in `Hashtable` and `Vector` than in `StringBuffer`. However, this prevention of data-corruption problems leads to harder to find logic errors. For example, here is a bug in Sun's own JDK 1.0.2 implemenation of `java.lang.String.intern`:

```
String s = (String) InternSet.get(this);
if (s != null) {
    return s;
}
InternSet.put(this, this);
return this;
```

`InternSet` is a `java.util.Hashtable`. The problem occurs if two threads try to `intern` the same `java.lang.String` at the same time. Both can probe and `get` the value and finding none, both will try to `put` their instances in as the interned value. This means that one of the callers ends up with a non-interned `String`. A higher level of synchronization is needed around the pair of `get` and `put` operations to prevent this. The prevention of the data-corruption problem masks the logic error of not producing interned `Strings`.

As in the `StringBuffer` case, third-party libraries provide non-synchronized versions of these classes, allowing the application to choose where it needed synchronization, instead of just paying it as a tax on general system performance. Netscape IFC provides the `netscape.util.Hashtable` and `netscape.util.Vector`.

Finally Sun's Java 2 version known as JDK 1.2 provided a new collection API allowing an application to choose between the older synchronized and the newer unsynchronized classes. These old and new worlds are unified through new List and Map interfaces implemented by both the old and new classes. Wrapper classes are also provided to turn unsynchronized classes into synchronized ones as needed.

However, there still is no unsynchronized alternative to `StringBuffer`, leaving that up to the application. This is unfortunate, since many APIs such as JDBC could benefit from taking `StringBuffers` instead of the usual `String` arguments, so the could reuse large mutable buffers, instead of allocating potentially large immutable `Strings` for every call.

Testing

One final word on synchronization is regarding testing. Based on the experience using Netscape IFC's unsynchronized classes, simple load testing finds synchronization problems quite readily. In a well structured application, there hopefully is not much global state to synchronize on, and where it does exist, hopefully the programmer got right the first time.

The reason it is easy to find the synchronization problems with the unsynchronized classes is precisely because it does lead to data corruption problems. Data-corruption problems end up looking very similar, usually a `NullPointerException` in a `Hashtable` or `Vector` read accessors or `IndexOutOfBoundsException` in a `Hashtable` or `Vector` write accessors. Given the Java stack trace it is easy to pinpoint the code that is lacking synchronization and which particular data-structured to which access needs to be synchronized.

In addition, since Java classes encourage encapsulation of such data structures, usually a small number of methods in one class are accessing the data-structure. At the very least, the code can be analyzed to find the users of the state to add protection that is needed, and any further stack traces found in testing can provide further leaks to plug.

7.2.3 Classes

Java would not be an modern object-oriented language without classes. As mentioned before, there is criticism of the lack of multiple-inheritance. This section will focus on other issues.

One problem with Java's class system is that it is not as dynamic as others such as the CLOS, the Common Lisp Object System, resulting in some non-objected-oriented approaches to some problems. One example is the static `Write.write` method. It has to do an `if/then/else` tree of `instanceof` operations to handle `java.*` and third-party classes.

It would be an interesting extension to allow dynamic extension of third-party classes to implement new interfaces with new methods.

One small nit with Java classes is that there is no clear way to have a class as a simple collection of static state. This is commonly done for sets of utility routines. A first thought would be to mark the class as `abstract` so that it cannot be instantiated. However, then a subclass can be created that can be instantiated. To prevent subclassing, the keyword `final` can be added to the class. However, Java does not allow the `abstract` and `final` keyword to be used together. In the end, the cleanest approach is to make the class `final`, but mark the constructor as `private` to prevent unwanted instantiation.

People seem to associate object-oriented programs with inheritance. However, encapsulation and interfaces are more important architecturally than inheritance. Encapsulation and interfaces allow for the reuse of whole packages where inheritance which is focused on the reuse of only a single class's implementation.

As packages are broken down over time to finer granularity, inheritance is often used to split the implementation into simple classes and their more complex superclasses. However, Java's single inheritance limits a class to a one-to-one relationship with its superclass.

Imagine that there is a class `A` that is now to be split into a superclass `B` and a subclass `C`. In the application as it stands today, there was one `A` so now there is one `C`, including its one `B`. However, as time goes on, suppose the application needs to have two `Cs`, but that automatically means there are two `Bs`, when perhaps one could be shared between the two `Cs`.

In the end, perhaps the more complicated component and interface model would have been better. Imagine that `A` had been split into a class `D` with a constructor taking an interface `E` and an additional class `F` that implements interface `E` with a constructor taking an `D`. `D` would be similar to `B` and `F` would be similar to `C`, with `E` defining precisely what behavior could be customized by users of `D`. Then if the application wants two `Fs`, they can share the single instance of `D`.

There are many such examples of "design patterns". Some of them involve inheritance, but mainly they involve interfaces between classes that are not based on inheritance. There is much more reuse to be had by pluggable components that simple subclassing. [15]

7.2.4 `RuntimeExceptions`

When programming in C/C++, the most common type of run-time errors were from problems in pointer arithetic and memory allocation, leading to segmentation faults, bus errors, etc. In Java, these problems have been replaced with `NullPointerExceptions` and `ClassCastExceptions`.

NullPointerException

`NullPointerExceptions` usually occur when a method has an argument or calls another method, but expects an honest-to-goodness `Object` reference back, not a `null` reference. The problem is that the `null` reference is considered assignable to any class. Other languages such as ML include whether or not `null` is allowed as part of the type, adding additional compile-time checking of values.

Often programmers return `null` when something unexpected occurs. Instead they should use exceptions, especially `RuntimeExceptions` like `IllegalArgumentException`, to signal the exceptional condition. When a large body of code simply returns `null` when an unexpected situation arises, several different methods may play the same game. By the time a `NullPointerException` actually occurs, the location reported maybe far away from where there issue was first detected. By eliminating such silent failures by throwing a `RuntimeException` where the problem first occured, subsequent debugging is much easier.

`ClassCastException`

Some object-oriented languages such as Eiffel have no casting whatsoever. However, since common data-structures such as `Vector` and `Hashtable` hold only `java.lang.Objects`, accessors of these structures must cast retrieved values to a more useful type. Eiffel and C++ solve this by having parameterized types or templates. However simple templating solutions can lead to code bloat.

One simple workaround is to have a `Vector`-like class that has abstract array allocation and array access routines. This allows the superclass to manage all the resizing and other bookkeeping. The subclass then provides strongly typed access to array elements. Although not as good as a builtin language solution, it does remove many chances of receiving `ClassCastException`, with a minimal amount of code bloat through shared implementation. In addition, access is faster than with a `Vector`, since once the array is retrieved, the only cost is for array access, without the additional method-call overhead of `Vector.elementAt`.

7.2.5 Assert and Macros

A common way to check for bad arguments such as null in order to avoid `NullPointerExceptions` is to use an assert mechanism. An example of defensive programming, a program may assert preconditions, postconditions, or invariants. The assert mechanism itself throws an exception when a condition is not met.

In C and C++, such asserts are usually enabled for internal builds but disabled in production products. This allows maximum validation internally, but maximum performance externally. However, this is implemented using the C preprocessor, for which there is no equivalent in Java. Eiffel does not rely on a preprocessor for its conditions but includes them as part of of the syntax of the language, allowing the run-time to disable them in a production environment.

Well designed macros would be a powerful addition to Java, especially if done as part of parameterized types. However, there are proposals for a simple assert mechanism, even declarative conditions, to help efficiently implement optional run-time condition validation.

7.2.6 `Numbers`

Java's `java.lang.Number` class is pretty thin. Although the arbitrary-precision subclasses `java.math.BigInteger` and `java.math.BigDecimal` come complete with methods `add`, `substract`, `multiply`, and `divide`, `Number` itself does not. They are not supplied statically by the `java.lang.Math` class either, leaving programmers to implement their own primitives to manipulate the Number classes. In addition, no builtin library for complex number support is provided.

7.2.7 `else if`

When a method throws an exception, the caller must choose to either throw or catch the exception. The caller can choose to catch the exception and do nothing to handle it, which some compilers warn about, but it is certainly an option.

However, a similar problem is not handling a branch in an series of `if/then/else` tests. Yale's T system provided versions of Scheme `cond` and `case` called `xcond` and `xcase` that would signal a run-time error if none of the branches was taken. A similar contruct in Java would be useful, and perhaps possible with a macro.

7.2.8 Exit

To exit the virtual machine, the `java.lang.System.exit` call is similar to the C `_exit` function. It immediately exits the process without any cleanup. However, most C programs call `exit`, not `_exit`, which allows exit handlers registered by `atexit` and `on_exit` to run.

Java lacks any standard way to allow code to cleanup on virtual machine exit. An application can have its own library `exit` method that does its own cleanup, however this does not allow third-party libraries to share one mechanism for cleanly shutting down. This means the application has to tie together all the third-party mechanisms in sometimes ad hoc ways.

7.2.9 Tail Recursion

Finally, Java as a language is lacking tail recursion. Even at the Java Virtual Machine level tail recursion is not possible, showing that the Java Virtual Machine is really not general purpose at all. Even the Gnu C Compiler, `gcc`, supports tail-recursive optimizations.

However, the IBM Java Virtual Machine's JIT compiler does in fact perform tail recursion elimination to cut down on method-call overhead. This is just another way that IBM has begun to edge out Sun. [58]

Outside the world of Java, Microsoft's Common Language Infrastructure's Intermediate Language does support tail calls. [12] There already is a Scheme system from Northwestern that is built on top of this platform. [62] It will be interesting to see if Sun decides to evolve the Java virtual machine in this direction or continues to focus one language for its virtual machine.

7.3 Scheme Advantages

This section discusses some of the advantages of using Scheme as an extension language to Java.

7.3.1 Size

The main reason Scheme was choosen was for the small size of its language. This few types of kernel synax and the uniform s-expression syntax allowed a small implementation to be up and running quickly. Although later the Scheme libraries were also implemented, they were not as important and added primarily for completeness. Most developers prefer to use the Java APIs over the Scheme versions.

7.3.2 Garbage Collection

Perhaps it goes without saying that garbage collection is an advantage of Scheme given that the implementation language Java is garbage collected as well. However, independant of the implementation details it is important for any scripting language to be garbage collected. Scripts are often written by inexperienced programmers and memory leaks are a very common type of mistake. If the application is a long running process, such as the server that was the embedding application for this implementation, leaks caused by user scripts could be very dangerous.

7.3.3 Functional Programming

Scheme functional programming style, which discourages side-effects, works well for embedding it in Java. The Scheme style meshes well with both multi-threading and transactional based systems.

Code with extensive use of side-effects does not work well in a multi-threaded system because of the overhead of the required synchronization. In addition, some simplistic libraries may assume they can side-effect a data-structure they are passed. Another problem is when a piece of single-threaded code reuses a data-structure within a loop, such as by clearing a Hashtable, to reduce allocation. Later on if the code is made multi-threaded, suddenly reusing the Hashtable does not seem like such a good idea.

Transactions and side-effects seem to be a better match. After all, transactional update is all about managing side-effects in a well-structured way. However, although the end results of a transactional computation are side-effects, it is good to avoid costly intermediate side-effects if they are not necessary.

Scheme avoids this by discouraging side-effects. For example, although Scheme includes a `reverse` function, no side-effecting `reverse!` function is included in the standard.

7.4 Scheme Disadvantages

Unfortunately, today Scheme seems to have less pros and more cons. A lot needs to be done to either modernize the language, or perhaps a new off-shoot of the language needs to be created to bring it up to par.

7.4.1 Language

This first section will focus on language, rather than library, issues.

Symbol

As mentioned above, the standard does not nail down performance behavior for `symbol` equality. It could be as cheap as a constant time comparison or the cost could be dependent on the length of the name of the symbol. Although this is a small matter and most implementations do perform as expected, in general the specification focuses on correctness more than performance, which is noble, but not practical.

Records

Since Scheme does not have a record system, programmers have tended to add their own, which leads to a proliferation of options. For example, `scsh`, the Scheme Shell, includes four different record systems. [53] [54]

The lack of a standard record system is unfortunate for many reasons. First, applications developers are forced to deal with what should be a language issue. Second, standard libraries are dumbed down to avoid using records. Third, each different extension library may have its own record system, incresing the learning curve for users of those libraries.

Finally, meta-level systems trying to provide record serialization or persistence have no general mechanism to rely on. This includes how the standard read and write procedures deal with records, which are often designed by a specific implementation to handle the system's preferred record system, but treat others as second-class citizens.

Related to the need for a standard record system is the need for concise syntax for manipulating records. Although Scheme programmers often criticize C-like languages for their variable and argument type declarations, most Scheme record packages end up including their type information in the name of the functions used to manipulate the variable. Special syntax for manipulating records could be used for clean integration with C `structs` as well as C++ and Java classes.

Types

As mentioned above, the Java based implementation was able to replace about a dozen type discrimination predicates with a single instanceof? function. If record types are added to the language, this will increase the importance of having a single function for type discrimination as a standard part of the Scheme language itself. Something as simple as a `type` function that returned a `symbol` such as `pair`, `char`, or `vector` would be sufficient. This would allow code currently explicitly testing for each type, perhaps for serialization, to use a table-driven approach instead. [1]

Threads

In today's world of multi-processor machines, languages must support threads. For languages like Java, they are perhaps to be considered almost a library feature. Scheme's concepts like `call-with-current-continuation` are not fully specified in a threaded environment. It would be better to include threads and thread-local storage, perhaps as dynamic variables, as part of the language.

[1] There seems to be an inconsistency in R5RS. "Section 3.2 Disjointness of types" mentions `port?` However, "Section 6.6.1 Ports" defines `input-port?` and `output-port?`, but not `port?`.

Exceptions

The Scheme standard uses the phrase "an error is signalled". For example, `open-input-file` can signal an error if a file does not exist. However, there is no function to determine if a file exists or to handle the error without terminating the program.

As mentioned above, exceptions are an important building block to enable modular libraries to come together seamlessly into applications. Without an exception system, libraries are tempted to try to handle exceptional cases, such as missing files, which are much better handled by the application which has a better understanding of the context in which the error occured.

Unforunately, Scheme, with its minimal static analysis, will probably never provide rigorous handling of exceptional cases, like that of the Java compiler. The lack of an object system also makes it more difficult to provide structured exception handling, requiring the application to exhaustively handle related problems, and be modified whenever new exceptional cases are added.

`call-with-current-continuation`

The Scheme standard notes the two common uses for `call-with-current-continuation`. The first is for non-local exits from loops or procedures, which is similar to Java's `break` and `return` respectively. The second is for escaping across several levels of a call stack, similar to Java's exception mechanism.

`call-with-current-continuation` is not limited to these escape-procedure contexts. Continuations are first-class procedures that can be used at any time and even multiple times to restart a computation. This has been shown useful for implementing cooperative threading where certain library procedures store the state of the current computation and switch to another pending computation. Certain programming techniques such as backtracking also are easily expressible using the `call-with-current-continuation`.

Many implementations do not properly implement the full power of `call-with-current-continuation`. This implementation only handles the common case of escape procedures. In general, implementing fully general continuations can be very expensive since the program stack may have to be copied to the heap if analysis could not show that the continuation would not escape the enclosing call to `call-with-current-continuation`.

One could argue that Scheme could be better off with specific constructs for specific features instead of one fully general `call-with-current-continuation`. With specific constructs for the useful concepts of non-local exit, exceptions, and threads, the more esoteric uses like backtracking algorithms would simply have to perform their own state management.

Modules

Scheme, as any language, requires a module system for two reasons. A module system prevents unrelated libraries from having namespace collisions. In addition, a module system allows a library to encapsulate its implementation and only export a defined interface.

Although some implementations do provide module systems, many do not. However, a standard module system is even more important than a standard record system. Although it is a nuisance for a programmer to have to learn several record systems when dealing with several libraries, without a standard module system, the libraries may not even be able to coexist.

`do`

`do` seems out-of-place with the rest of Scheme. While Scheme has a Lisp heritage, it seems like Scheme's focus on looping through tail recursive function call makes the `do` syntax redundant. Although this non-tail-recursive implementation used `do` extensively in implementing standard functions, it seems out-of-place in an otherwise clean language.

N-ary Arguments

N-ary arguments seem like another piece of baggage from Scheme's lisp heritage. N-ary arguments require list allocation in order to pass their values, which is not a great thing to encourage for performance. It also leads to complicated APIs with perhaps mutliple levels of implicit defaults instead of an efficient and clear API.

Static Analysis

Scheme as a language promoted lexical scoping over dynamic scoping. One benefit of lexical scoping is that it allows for compile-time optimizations, such as the closure analysis shown above. Scheme also distiguishes the compile-time `'(1 2 3 4)` from the run-time `(list 1 2 3 4)`, going so far as to note that side-effecting the former is an error.

However, Scheme as a language otherwise does very little to allow other forms of static analysis. Since it has very general arithmetic and allows users to replace the definitions of standard functions, many forms of optimization are off limits. Many implementations include declarations to allow compilers to perform more performance optimizations. However, few include include declarations for assisting in program correctness.

Scheme 48's module system allows for function signatures which include type information. Although the type information seems to be informational, the compiler does at least warn if a function is called with an incorrect number of arguments. One could imagine that some simple analysis could be done to at least detect some type incompatibilities. [29]

7.4.2 Libraries

This section address Scheme's short-comings in the area of standard libraries. For a language including `transcript-on` and `transcript-off` as optional procedures, there certainly are a number of more useful things that could be included.

Data-Structures

Scheme is lacking a concept of date and time. In this implementation, extensions for `java.util.Date` were available and used to write benchmarking code. A good date data-structure would probably rely on a standard record system. Dates will also appear as part of I/O and internationalization libraries below. Time zone support would be considered part of a core date library, not part of internationalization.

Hashtables are another incrediblely useful data-structure. Most Scheme implementations provide hashtables, but having a standard one would improve portability of libraries and applications.

Scheme vectors are similar to Java `Object` arrays. However, Scheme provides no equivalent to Java's `System.arraycopy`. Certainly a program can iterate over a vector copying elements, but there is a performance improvement from providing it as an atomic operation with a custom implementaion.

Since Scheme `vectors` are similar to Java arrays, an equivalent to Java `Vectors` would be nice. Scheme programmers would tend to use list structure where Java programmers would use `Vectors`. However, accessing elements in Scheme is a linear operation. If `list->vector` is used to convert to a constant-time access data-structure, it means the `pairs` were allocated only to become garbage.

Batch

Scheme is lacking the basics needed to operate as a batch program. Although some of the first things any C programmer learns is how to use command-line arguments and how to return an exit code, Scheme provides no standard functionality for either of these. Almost all implementations do, since it is useful on almost all common systems. One could argue that it presumes some specific type of operation system, but the presumption of a filesystem already means that not all library procedures may be available in all environments, as seen in the Java `Applet` environment.

As mentioned with Java, any exit facility should provide exit hooks to applications and libraries. One useful facility that is useful for batch programming that needs such an exit hook is for temporary files, to ensure they are cleaned up on program exit.

Properties

Most language systems allow access to named string properties. C provides `getenv` and `setenv` to access environment variables. [2] Java's System Properties are incredibly useful for the unfortunate but necessary times when a program needs to vary its execution based on its architecture, operating system, virtual machine, language version, etc.

Scheme programs should have a standard way to differentiate between Scheme implementations. This would go a long way to allowing programs to create their own portability libraries for non-standard features until such a time that they are standardized. For example, an application could provide their own implementation for accessing command-line arguments to hide the details of the implementation. Just as easily, they could even build a portable record system or hashtable implementation.

[2] Java JDK 1.0 had `getenv` and `setenv`, but they were removed in JDK 1.1 because the concepts did not port to some environments like the Macintosh. Ironically the `Runtime.exec` method still provides an `envp` argument for passing environment variables to subprocesses.

Miscellaneous

Other common functions such as `any?`, `every?`, `fold`, `reduce`, `reverse!`, etc. would be useful to have provided in some standard library.

7.4.3 I/O

Scheme's standard I/O functions are so bad that this section is dedicated to just that part of the library. With redundant functions such as `with-input-from-file` and `with-output-to-file` included in the standard, it seems like more attention should be paid to what is omitted.

Streams

The inclusion of `peek-char?` on `input-ports` makes it clear that the Scheme standard authors are more worried about writing lexers than more general programs. Java's basic `InputStream` API focuses more on the essentials and layers on more complex behavior such as peek ahead.

One of the biggest omissions from the Scheme I/O library is the buffering semantics of ports. As far as the specification is concerned, there is a one-character buffer for `peek-char?`. However, if Scheme systems were really doing character-at-a-time I/O performance would be abysmal.

Scheme could benefit from a more extensible streams-like API. It would need to have a more object-oriented approach allowing each type of stream to supply its own implementation of operations such as read and write. This could easily be done with standard record for streams if a standard record system existed.

`transcript-on` and `transcript-off`

It is hard to imagine how something as pedantic as `transcript-on` and `transcript-off` ever made it into the standard. It would be more useful to provide direct access to set the current-output-port. Then if I/O streams were available, something similar to `MultiPrintStream` could be built. These two combined, with a little help with from the `REPL` could provide more useful functionality to professional programers, rather than primitives for students to generate files to turn into their professors.

Internationalization

As mentioned above, much of the work of internationalization is properly differentiating characters from bytes and providing the means to convert back and forth between the two. Scheme currently does not have a concept of byte, but that is not necessarily a major hinderance. A stream API could layer a multibyte Unicode character stream on top of a 8-bit character stream to simulate byte operations. However the standard could encourage implementations of `char->integer` to return a value matching the underlying representation whether that be ASCII or Unicode, since some implementations choose to return somewhat arbitrary values.

Internationalization also affects the character functions such as `char-whitespace?`, `char-lower-case?`, `char-upper-case?`, `char-numeric?`, `char-alphabetic?`, `char-upcase` and `char-downcase` as well as the string functions potentically built on top of them, such as `string-ci>?`, `string-ci<?=`, and `string-ci>=?`. For the ASCII character set, these operations are relatively straightforward. However, for Unicode, these functions require more complicated table-driven logic as well as many special cases, as well as ongoing maintenance to support the Unicode standard as it evolves.

In addition, if dates are provided as a standard data type, internationalization needs to include methods for parsing and formatting date objects for different international locales.

Portability

Scheme's `open-input-file` and `open-output-file` and related functions take strings to represent files. Java uses `Strings` as well, but most code uses the more portable `java.io.File` class. Common Lisp provides much more support for portable file operations.

Scheme would benefit from a file type along with a library of routines for the creation and manipulation of files, dealing with such portability issues as file-separator characters. In addition, a record type for file information would be useful. This record of file information could utilize a data type to report the relatively portable concepts such as last modification time.

Scheme could do without the concept of a current working directory, as Java has. As mentioned above, current working directory can be tricky to implement in a multi-threaded environment.

Network

Java is arguably the first language to include the concept of URL from its inception. Nowadays, URLs are expected to be integrated into any I/O system. In addition, traditional socket interfaces are necessary. Such libraries should make functionality like the `REPLServer` available as portable Scheme.

7.4.4 Platform

Scheme environments do not provide a very consistent platform for Scheme application developers. Before the implementation was complete, several different versions of Scheme were needed for Win32 development. Scheme 48 did not work on Win32. PCScheme was used for its debugger. mzscheme was used for its performance. mzscheme could not even provide a stack trace where the problem occured, and yet the authors did not seem to understand why that was frustrating. Eventually this implementation replaced the third-party tools but Scheme environments need to be more supportive of their users.

7.4.5 Testing

The Scheme standard is known for its formal language semantics. While many Scheme implementations correctly implement most of the language, somehow many implementation specific problems still arise. What is needed is a standard suite of tests to clarify subtleties from the language specificication as well as provide a sanity check for implementors.

There are two categories of problems to test for. The first category of problems are language compability issues. The second category of problems are library issues. Given the small library, writing a comprehensive test should not be difficult.

For language compability, some of the problems are small, for example syntax errors that are harmlessly tolerated in one implementation that lead to portability problems in other implementations. Another example is supporting little known syntax like `=>`. However any implementation limitations for constructs like `call-with-current-continuation` can also lead to subtle portability problems.

However, it is not easy to write a Scheme program to detect syntax errors. One problem with the implementation of the `Let2Application` rewriter function was that it tolerate the following syntax error by simpling ignoring `bar`:

```
(let ((a "foo" "bar")) a)
```

If Scheme had a standard exception mechanism, it could try to load this bad syntax and make sure that the implementation signalled an error as expected. Manual inspection after this problem was found led to many other problems with syntax rewriters being plugged, showing their error prone nature.

In the area of library testing, Aubrey Jaffer's test.scm caught many small issues after the implementation had been in use for some time. However, since test.scm restricts itself to using only standard Scheme in its implementation, it is not able to do negative testing. [25]

7.4.6 Goals

A short-term goal for the evolution of Scheme would be to extend the language and libraries to the point where most Scheme compilers could be written entirely using the standard libraries. A longer term goal would take the evolution a step further to enable the construction of a well performing multi-user database system. Perhaps such accomplishments would inspire new generations of programers, moving Scheme out of its place as a language for computer-language theorists.

Chapter 8

Language Discussion

This section is for general discussion of programming language issues raised in the work on this implementation. Although much of the discussion centers on Scheme and Java, perspectives from other languages are also incorporated.

8.1 Code-Data Duality

Scheme, and Lisp systems in general, are visually distinct from other programming languges because of their s-expression syntax. However, looking back at the evolution of this Scheme system it was interesting to note that it was not until after the `REPL` was fully up and running with a working kernel language, that traditional pair primitives such as `cons`, `car`, `cdr`, `list`, etc. were added. As in SICP, it was even longer before functions to side effect pairs were added. It was even later still before user-definable rewriter macros were added, where the code-data duality is perhaps most visible and important.

Scheme as a language is too closely tied to its list-processing heritage from Lisp. Most users of this scripting system did not care much about performing list operations, since the data strutures they manipulated were Java based.

Scheme's mapping of code into poorly typed list structures could be done in a different way. Scheme compilers internally do not usually choose to represent code as lists. They usually represent code with a syntax tree of record types. What would a Scheme macro system be like if there were a standard Scheme record system and there existed functions to map standard syntax records to and from s-expression syntax. Certainly it seems like this could make life easier and less error prone for macro writers.

Taking this idea to Java, it might be possible to take this language without preexisting code-data duality and perhaps gracefully add it. It is easily imagined that Java classes could be defined to represent the structure of the Java program itself, certainly the javac compiler, itself written in java, internally has a representation of this sort. javac even indirectly provides a view into this representation via the Doclet API that allows the `javadoc`— API documentation generation tool to allow user code to inspect a subset of this representation at the package, class, and member level, although not down to the statement and expression levels.

Java is currently missing the pieces to tie this together into a useful macro system. Certainly it is good that JavaSoft has kept `cpp` preprocessor style macros out of Java, but having nothing has been limiting. It would be an interesting project to try and build a modified Java compiler that would provide explicit rewriting style macros, as well as perhaps more advanced R5RS style macros, or even macros that allow static type checking so errors could be reported in terms of the programmers unexpanded code.

8.2 Packages and Modules

All in all, Java packages do fufill the two basic goals of a module system: namespace cleanliness and implementation hiding. Other languages have different takes.

Perl packages use nested namespaces mapping to directories and files which is similar to Java. The Perl language exposes the mechanism of how namespaces work though data-structures. This means that package imports and exports can be implemented in Perl itself, which is provided through the `Exporter` package. This means the package boundaries are not strictly enforceable, since any program can use the same mechanisms used by the `Exporter` package.

However, this can be good, because packages that need to bend the rules can bend the rules. Because the `Exporter` package works by accessing subroutines defined by a package, a package can programatically decide to export different definitions conditionally. This allows packages to provide more exports to related packages, similar to C++ friend classes. In fact, packages are so flexible, they are also used as the basis of Perl's object system. Unlike Java, where a package and class are distinct concepts, in Perl they are one in the same.

Microsoft's new C# language criticizes Java packages for being too tied to the directory and file name of the source file. Actually this is not strictly necessary, in fact most compilers only warn if the filename does not match and none appear to constrain the directory name. The output class file does always follow the convention. [11]

C# allows the `package` namespace declaration to be used at any top level context. Multiple namespaces can therefore be freely manipulated within the same file. Perl actually allows similar use of the package keyword and it primarily is used to define helper classes within the same file as the main exported class, not to haphazardly mix namespaces. So although C#'s `package` declaration is not unlike Perl, it does not seem an improvement over Java, where, unlike Perl, there are distinctions between packages and classes. In fact, indiscriminate use of this extension hurts both humans and tools in their ability to automatically locate the source code of the class based on solely the class name. A debugger might be able to pull this from debugging information, but to a human reader the package name to file name convention is useful.

8.3 Type Safety

The C and C++ programming languages have the concept of a `void*` pointer which is a pointer to any data type. Java's equivalent concept is a `java.lang.Object` reference. These untyped pointers or references are useful for generic data-structures as well as to provide application specific context information in call back APIs. C, C++, and Java rely on type casts to convert from these untyped references to more specific types.

One of Pascal's historic limitations was that it's required strict compile-time type saftey, without any run-time type casting. This often meant duplicating code for each record type to support linked lists or other data-structures.

Eiffel also has requirements for strict compile-time type safety with no type casting. However, since Eiffel does offer parameterized types, it does allow generic data-structures. Unfortunately many of Eiffel's generic collection types are not multi-thread safe. Unlike Java that separates the iteration state into `Enumerations` separate from objects like `Vector` and `Hashtable`, Eiffel's library classes keep iteration state in the `Object` itself, preventing multiple threads from iterating through an `Object` simultaneously.

C and C++ type casting allows potentially unsafe casts between any values. `void*` pointers can even be cast from pointers into types such as `int`. This is useful to the language implementor for performing pointer arithemetic to implement tagged pointer values. Java does not allow such games, only allow safe run-time checked `Object` casts or numerical casts, but casts between the two categories are not allowed. While this type safety is good for the application programmer, it is inconvenient for the language implementor.

8.4 Dynamic Invocation

Most language systems have a form of dynamic invocation. Most C language systems allow a program to look up a function pointer from a symbol name although the signature of the function pointer has to be known at compile time. Scheme's `eval` allows a program to dynamically create and invoke an s-exp. Java's reflection allows a program to dynamically enumerate the members of a class and then access fields and invoke methods that were unknown at compile time.

8.4.1 C

The C method is very efficient. Once a pointer has been looked up from a symbol, the cost of invoking the function is the same as invoking a function known at compile time. Although the function signature of a dynamically invoked function needs to be known in advance, this is still generally useful. For example, the Scheme 48 system's foreign function interface uses this signature for external functions:

```
long f(long nargs, long *args)
```

This signature is very similar to the `applyN` signature used in this implementation for its Java defined foreign functions. Similar to how primitive `Procedure` subclasses perform type marshalling and then call a standard Java library, Scheme 48's external functions usually use a library of Scheme 48 code to perform marshalling and invoke C libraries. [29] Tcl uses a similar interface to integrate to C.

However, as with writing primitives in Java, most of the code often is boilerplate. To avoid the tedious handcoding of wrappers, Cig, a C Interface Generator, provides a declarative way to define interfaces from Scheme 48 to C functions, providing code generation for

C stubs that implement the external function signature. [52] In this way Cig provides the analogous functionality to Scheme 48 that XS provides to Perl.

8.4.2 Scheme

Scheme's `eval` is another form of dynamic invocation. Scheme's `apply` is closer to what C offers. What makes `eval` different is a program is created from a s-expression data-structure dynamically at run-time. This would be like a C program creating a `char*` that contained some C code and then compiling it and invoking it. Similarly in Scheme, this implies the presence of a compiler to process the source to be evaluated.

Although a simple s-expression interpreter has no compiler to speak of, most Scheme systems have some sort of compiler. If a program uses `eval`, then that compiler has to be around at run-time, not just compile-time, bloating the run-time footprint. Some systems, such as Kawa, provide a simple interpreter to avoid the cost of always using the compiler. However the footprint problem does not stop there. A Scheme compiler could aim through static analysis of a complete program to package a minimal run-time system that include only the needed libraries. But if a program uses `eval`, there is no static analysis that can be done to create a minimal execution environment.

8.4.3 Java

Java's reflection is a mixture of what is found in C and Scheme. In Java only existing classes can by dynamically invoked, similar to C, and avoiding providing a Java to byte-code compiler in the run-time system. However, unlike C, the signature of the methods does not have to be known at compile-time. If the signature was known at compiler-time, then `Class.newInstance` followed by a cast to the compile-time signature would be sufficient. Unfortunately, as shown above, the cost of using dynamic access is very expensive when compared to normal non-reflective access.

One ability that Java reflection provides that is not found in standard C or Scheme is the ability to enumerate through all the packages known to the system, all the classes in each package, all the members of each class, and finally the signature of each member. Although some C run-time systems do allow a list of symbols to be returned from a library and perhaps some Scheme module systems allow their signatures to be analyzed, neither seem to provide full signature information for the exported functions.

Finally, all dynamic invocation tends to involve mapping string or symbol names into something that can be executed. In Java, as in C, this lookup happens once, and either suceeds or fails at that time with a `ClassNotFoundException` in Java or pernaps a null pointer in C. Unfortunately the Scheme standard does not define what happens if there is a problem, such as an undefined variable, in code dynamically invoked via `eval`. It is not even defined to signal an error. A program needs to be able to use `eval` with possible error-ridden user-supplied code but gracefully recover. Without some sort of exception system, there is no portable way to do this.

8.5 Threads, Dynamic Variables, and Thread-Local Storage

Scheme's major break from Lisp was its use of lexical instead of dynamic variables. However dynamic variables have their place fulfilling the role of "global" state for concepts like the current session or current transaction. Dynamic variables are useful today because they provide a form of thread-local storage if implemented correctly. For Java, JDK 1.2 provides `java.lang.ThreadLocal` as mentioned above. It also provides the twist of `java.lang.InheritableThreadLocal`. This provides defaulting thread-local values for new `Threads`.

`InheritableThreadLocal` might seem to provide the necessary support for dynamic variables for a Scheme in Java implementation with threading support. However the semantics of dynamic variables in a threaded environment are not clear. Assuming that a new thread inherits its parent's dynamic variables, what are the semantics when a dynamic value is assigned in the old thread? One possibility is that the new thread sees the new value. A second possibility is that once the new thread has been created that the dynamic variables could be modified independently, with perhaps copy-on-write sharing going on underneath. `InheritableThreadLocal` actually shares `Objects` by reference which means that immutable classes have the bevahior of the second possibility. However, if a mutable class such as a `Vector` or `Hashtable` is shared, the behavior is more like the first possibility. Fortunately, a program can subclass `InheritableThreadLocal` to provide copy semantics if desired.

8.6 Syntax

One of the reasons Scheme was choosen as extension language for Java was because of its small, simple syntax. Java itself was seen as an improvement over the tangle of syntax that C++ had become in adding object-oriented programming.

One bad aspect of macros is that by allowing developers to create their own syntax they can often just make programs less readable. Macros are only syntactic sugar anyway, and as Alan Perlis said, "Syntactic sugar causes cancer of the semicolon."

Java has started down the slippery slope of adding new syntactical sugar. While some conveniences such as array literals and `Class` constants are useful, more complicated syntax such as inner classes and anonymous classes do not seem to add much value. Java needs to consider truly new functionality like asserts or parameterized types, not simple sugar for existing functionality. Needless new syntax seems to be setting Java down the road of Perl which prides itself on its syntactical shortcuts and its infinite variety of ways to peform simple tasks.

Chapter 9

Comparative Analysis

In addition to analyzing the implementation itself, it is helpful to perform a variety of different types of comparative analysis with other similar systems.

9.1 Comparative Analysis with other Scheme systems

This section will compare the implementation with other Scheme systems. A short overview will be given of each system followed by some performance analysis.

Lisp systems have historically been performance tested using the Gabriel benchmarks originally written by Richard P. Gabriel. Will Clinger ported these benchmarks to Scheme which are available from the Scheme repository. [10] Clinger's version is incompatible with the current Scheme standard where null and #f are distinct objects. Fortunately Jeffrey Mark Siskind updated these for more modern Scheme system and distributes them with his Stalin system. [56]

The raw data for the performance results is included in Appendix A. All tests were performed on a Dell Dimension T800r system with an 800MHz Intel Pentium III processor with 512MB of RAM running Windows 2000 Professional Service Pack 1.

See Appendix A on page 159 for the raw results.

9.1.1 Java Scheme Systems

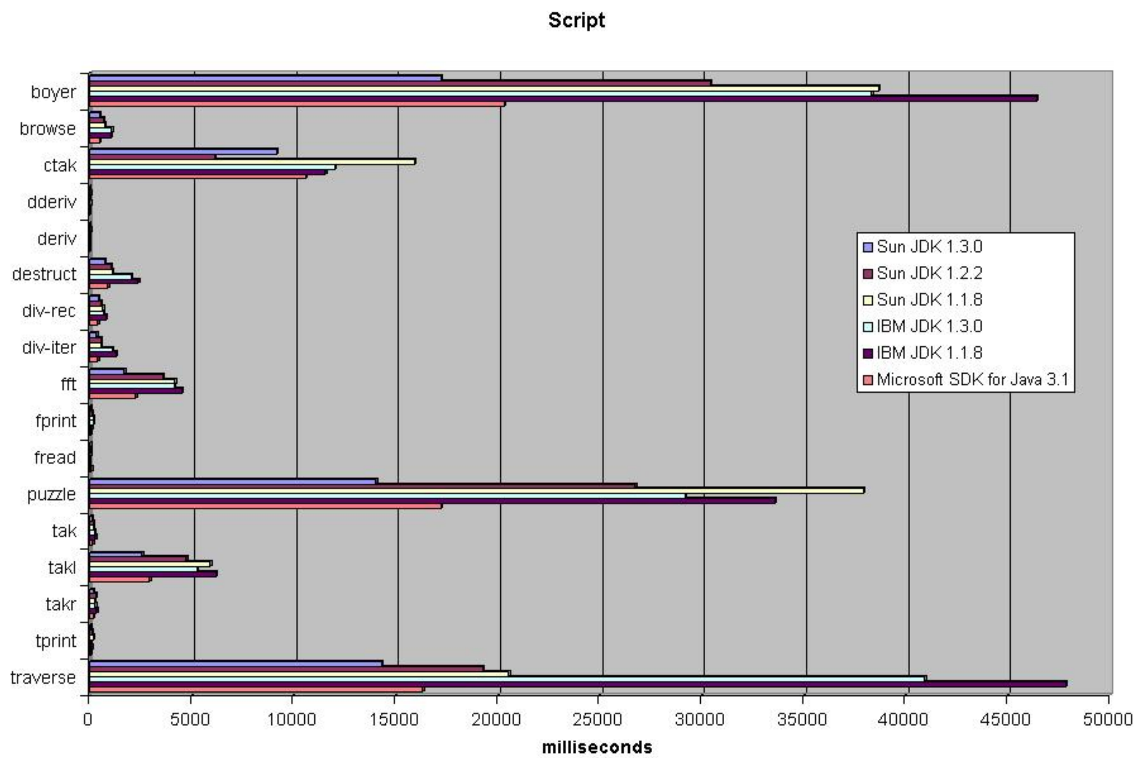
This section focuses on comparing with other Java-base implementations. One important consideration for testing and comparing Java programs is to test against a variety of Java virtual machines. Most of the Scheme in Java implementations now require the widely available JDK1.1 from Sun or a compatible implementation such as Netscape's. However, Sun's JDK1.2 and later virtual machines have much improved run-time performance. In addition to Sun's reference implementations, Microsoft's SDK for Java and IBM's JDK have different performance characteristics because of alternative foreign-function interfaces, data representations, and garbage collectors.

For all Java-based implementations, results are shown for several common virtual machines:

- Sun JDK 1.3.0
- Sun JDK 1.2.2
- Sun JDK 1.1.8
- IBM JDK 1.3.0
- IBM JDK 1.1.8
- Microsoft SDK for Java 3.1

Script

The first results to present are for the Script implementation.



As expected the newest Sun and IBM virtual machines improve on the performance in most cases. One surprise is how poorly the IBM virtual machines perform compared to those from Sun, given its reputation for out-performing Sun. Another surprise is how well the Microsoft virtual machine performs. Only Sun's latest offering beats the somewhat dated Microsoft implementation.

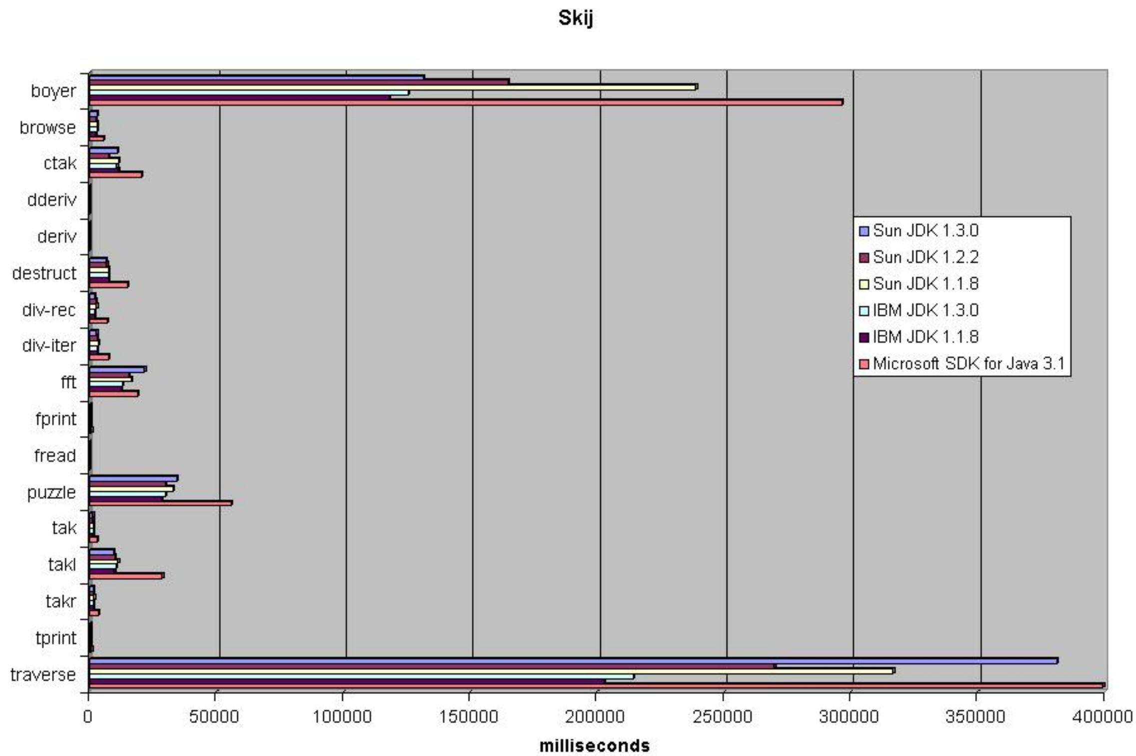
Skij

Skij by Michael Travers is available from IBM alphaWorks. [63] Skij provides `Applet` support as well as a console interface. Skij can be embedded in an application, but only can have one interpreter per virtual machine.

Skij deviates from Scheme in several ways. Symbols are case-sensitive as with the Script implementation. `string-set!` is not provided because only immutable `java.lang.String` instances are used to store Scheme `string` objects. `call-with-current-continuation` is limited to escape procedures as with Script. Skij is not tail-recursive. Extensions are provided for reflection, exception handling, dynamic variables, defmacro, and event callbacks from Java-to-Scheme.

One interesting feature of Skij is its Swing inspector. This allows any Java object to be browsed in a graphical window. The object currently being inspected is available to the interpreter through a global variable allowing the application to select the object to inspect.

Skij version 1.7.3 was used for running the benchmarks. A missing two-argument version of the `atan` function was added for running the `fft` benchmark using `java.lang.Math.atan2` and Skij's reflection API.



Skij does not do nearly as well on the Gabriel benchmarks as the Script implementation. The most obvious cause is that the implementation interprets an s-expression tree directly. In addition, while the global environment is stored in a simple Hashtable, lexical environments are stored using association lists.

Unlike the Script implementation, IBM's Skij shows an improvement on the IBM virtual machines. This time Microsoft's virtual machine does not fair as well in general. One big surprise is that the 1.3.0 virtual machines show a performance degradation over the earlier release from the same vendor.

SILK

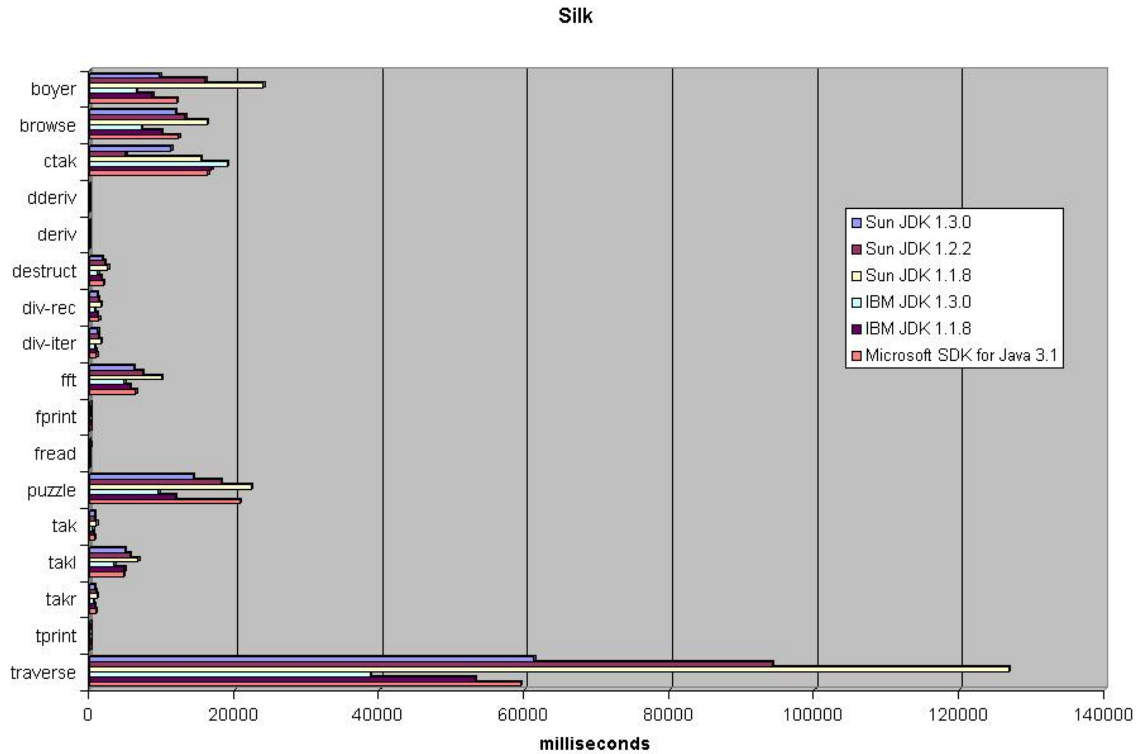
SILK started as a small Scheme in Java system by Peter Norvig. [46] It merged with Tim Hickey's JScheme, where it picked up its JLIB, its `java.awt` library. [21] Today SILK is maintained by Ken Anderson, Tim Hickey, and Peter Norvig. SILK provides an applet environment as well as console mode. It can be embedded in a Java application but allows only a single interpreter per virtual machine. The Java API is more fully-featured than Skij, but throws `RuntimeExceptions` instead of including a declared `Exceptions` in its method signatures because the authors, as they describe themselves, are "lazy". SILK packs most primitives in one large class instead of having one primitive per class to cut down on on the runtime footprint of the application.

SILK originally used `char[]` to represent Scheme `strings` but switched to the immutable `java.lang.String`. `call-with-current-continuation` is limited to escape procedures. SILK is not fully tail-recursive, but does some analysis to support self tail-calls. SILK has extensions for reflection. There is special reader syntax for reflection to make it less intrusive. SILK's reader started with StreamTokenizer, as did the Script implementation, but later it was thrown out.

SILK has a Scheme-to-Java compiler. However, this is not a sophisticated byte-code compiler, but really serves as a form of serialization. The resulting Java class can be run directly or loaded into an interactive interpreter. This allows the standard functions that are defined in Scheme to be compiled into a Java class, allowing the runtime to include only Java class files, without need for Scheme source files.

SILK also has a console-based `describe` for browsing Java objects, similar to Skij's `inspector`.

SILK version 4.0 was used for running the benchmarks. It was also missing an two argument version of `atan` as well as a two argument version of `make-vector` and `fill-vector!`. SILK documents the issues with `make-vector` and `fill-vector!`, noting these are optional procedures, but they are needed to run the Gabriel benchmarks.



SILK does much better than Skij on the Gabriel benchmarks. Script does tend to do better, although Silk wins on `boyer` and ties on `puzzle` for the Sun virtual machine. Since its early implementation SILK has added many of the optimizations found in the second-pass implementation of Script, but apparently none from the third pass.

The IBM virtual machine shines for SILK, giving SILK the lead on `puzzle`. Microsoft's virtual machine also makes a decent showing, beating Sun's 1.2.2, although falling behind IBM's 1.1.8. The SILK paper contains some performance benchmarking with Sun and IBM virtual machines against Guile, which is based on SCM system shown below. [4] [36]

Kawa

Kawa was original written by R. Alexander Milowski but has been rewritten by Per Bothner. [7] It has a console interface but also supports user interfaces, including JEmacs, a Java based Emacs implementation. Kawa can be embedded in an application and can compile Scheme modules into Java classes as well.

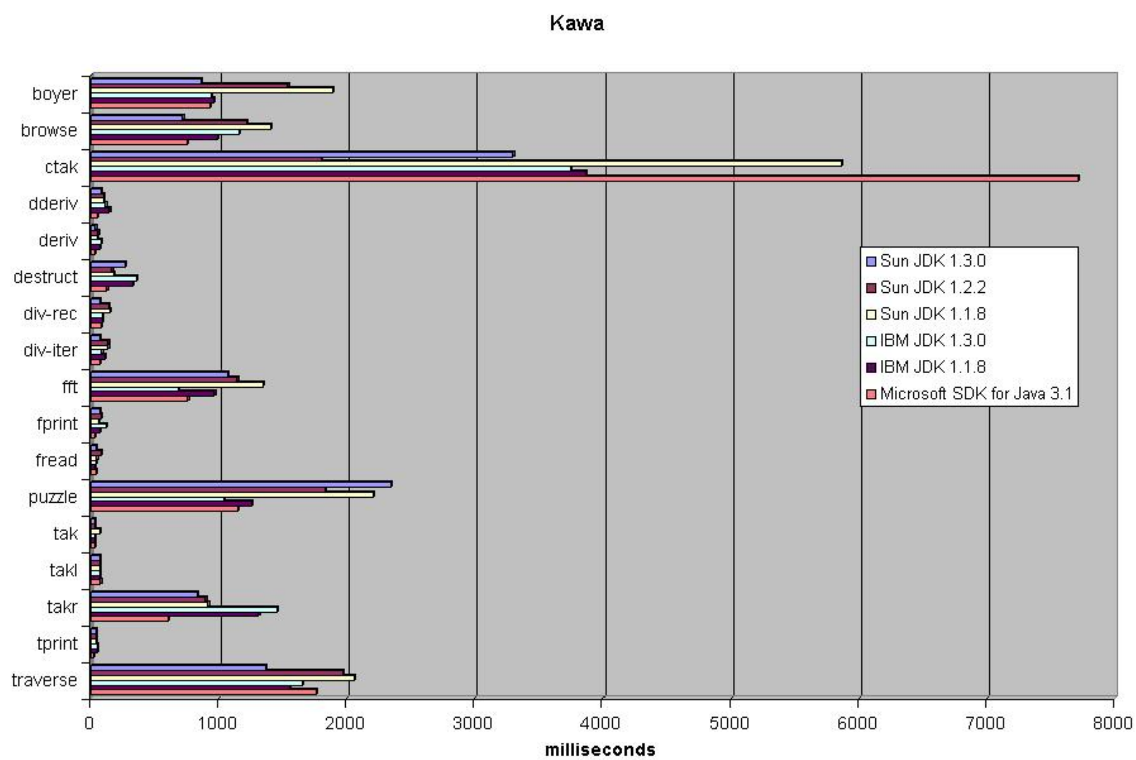
Scheme `symbols` are represented with Java `Strings`. In a deviation from most Schemes in Java, all other types are Kawa classes, including `vector`. By default tail recursion support is limited, but there is an option to fully enable it, although it encurs a peformance penalty.

`call-with-current-continuation` is limited to escape procedures.

Kawa has a large number of extensions. For Java, it includes reflection, exceptions, threads, synchronization, vector append, and `instanceof`. For Common Lisp it includes lvalues, formatting, and keyword arguments. For Scheme it provides records, dynamic variables, SRFI-4 for uniform vectors, and SRFI-6 for port operations. It allows optional type declarations in `let` and `lambda` as in RScheme. [31] It includes process extensions as described above. It provides an enhanced file system interface. It provides a Guile and scsh compatible `read-line`.[36] [53] [54] Since Alex's original implementation, Kawa has supported extensions to numbers for quantities and units to support DSSSL. Finally logical bit operations, extended string case operations, and generic functions are supported.

The most impressive feature of Kawa is its Java byte-code generation. It can compile a module of Scheme code to a Java class that can then be invoked by a Java program, or even act as a standalone Java application of `Applet`. For JEmacs, Kawa also is working on support for elisp and some Common Lisp. It also supports name properties on procedures, similar to the Script implementation. It comes with a regression test suite.

Kawa version 1.6.70 was tested. One patch was required from Per to fix a byte-code generation bug. `--full-tail-calls` was not used in running the tests.



Kawa takes the overall crown for the Java-based Scheme systems. This is almost certainly do to its byte-code generation. One detail to note is that it special-cases the application combinations involving zero to four arguments, as mentioned in the Script implementation above.

The differences in virtual machines is least noticable with Kawa. IBM and Microsoft beat Sun in `puzzle`. However Sun beats IBM in most other tests except `fft`. Some tests run slower in the 1.3.0 virtual machines, although in general the newer systems are faster.

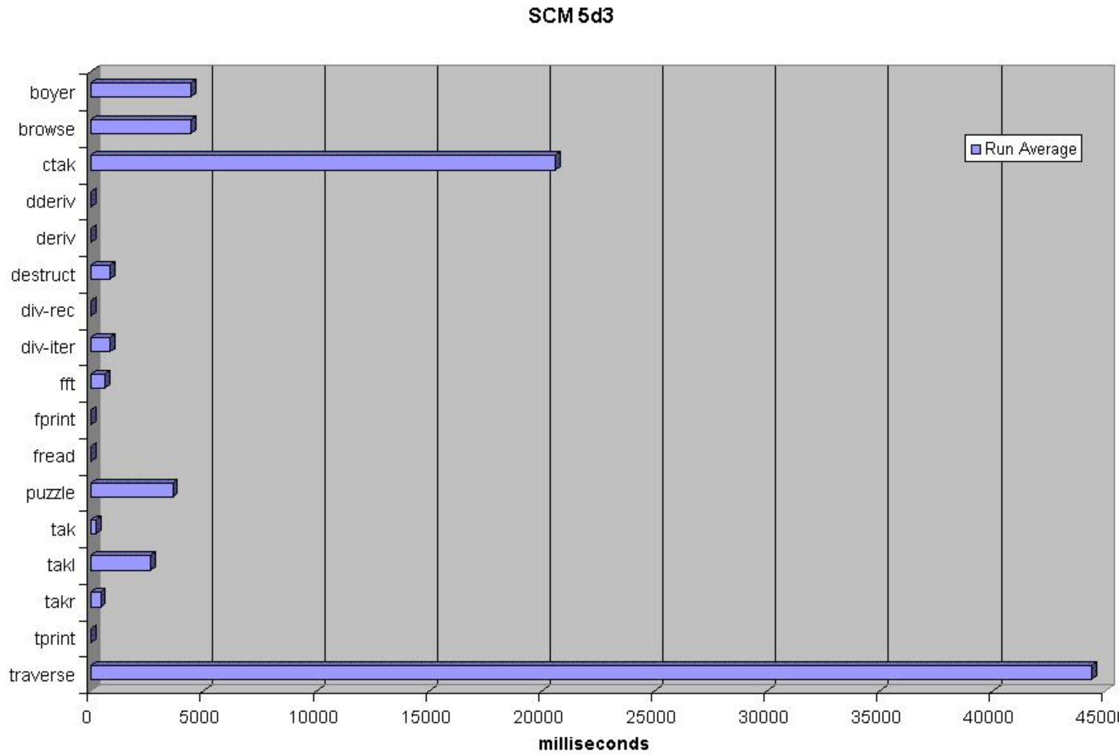
9.1.2 Non-Java Scheme Systems

This section covers several non-Java Scheme systems focusing on their performance on the Gabriel benchmarks.

SCM

SCM is very portable interpreter from Aubrey Jaffer that is available for a wide variety of operation systems. [24] As mentioned above, SCM is the Scheme implementation used in the Guile system.[36]

SCM 5d3 was used for benchmarking. SCM's timer granularity seems to be seconds not milliseconds so the numbers are not an exact match against the other systems.

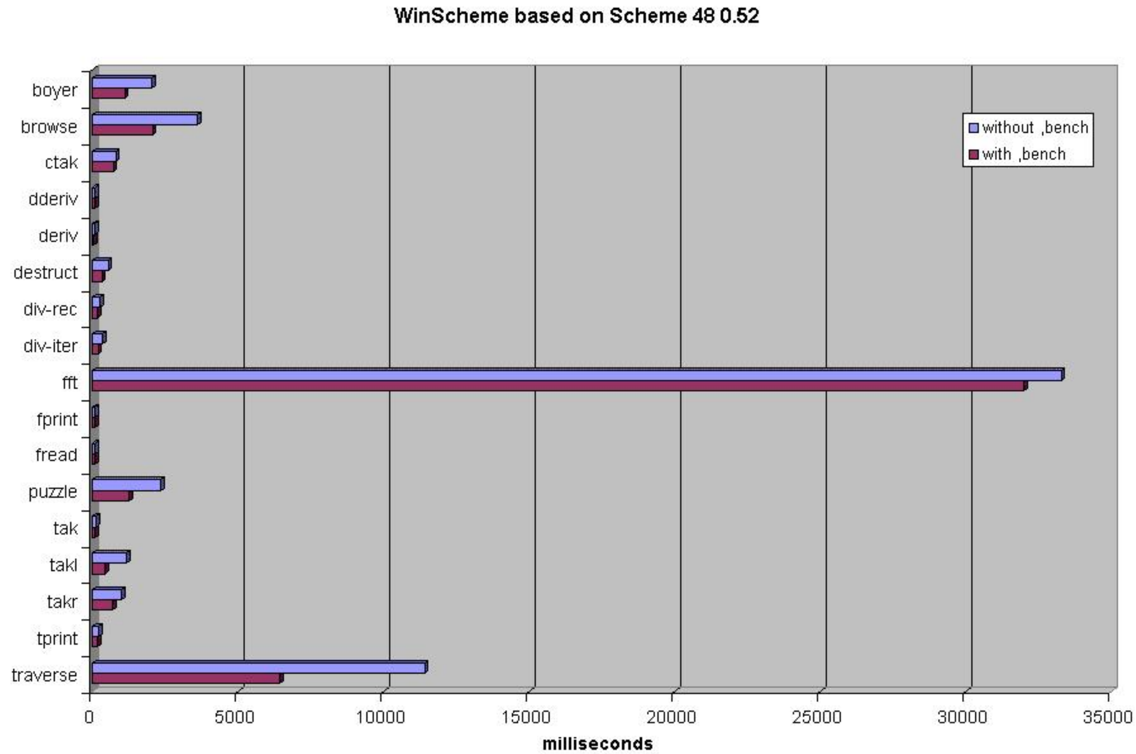


Script stacks up respectfully to SCM. SCM does beat it handily on some tests such as `boyer` and `puzzle`. However, Script does very well on `browse`, `ctak`, `traverse`.

Scheme 48

Scheme 48 is the product of Richard Kelsey and Jonathan Rees. Scheme 48 differentiates itself from most other Schemes through its byte-code virtual machine. [29]

WinScheme48 based on Scheme 48 0.52 was used for testing. The tests were run both with and without the `,bench` benchmarking option.

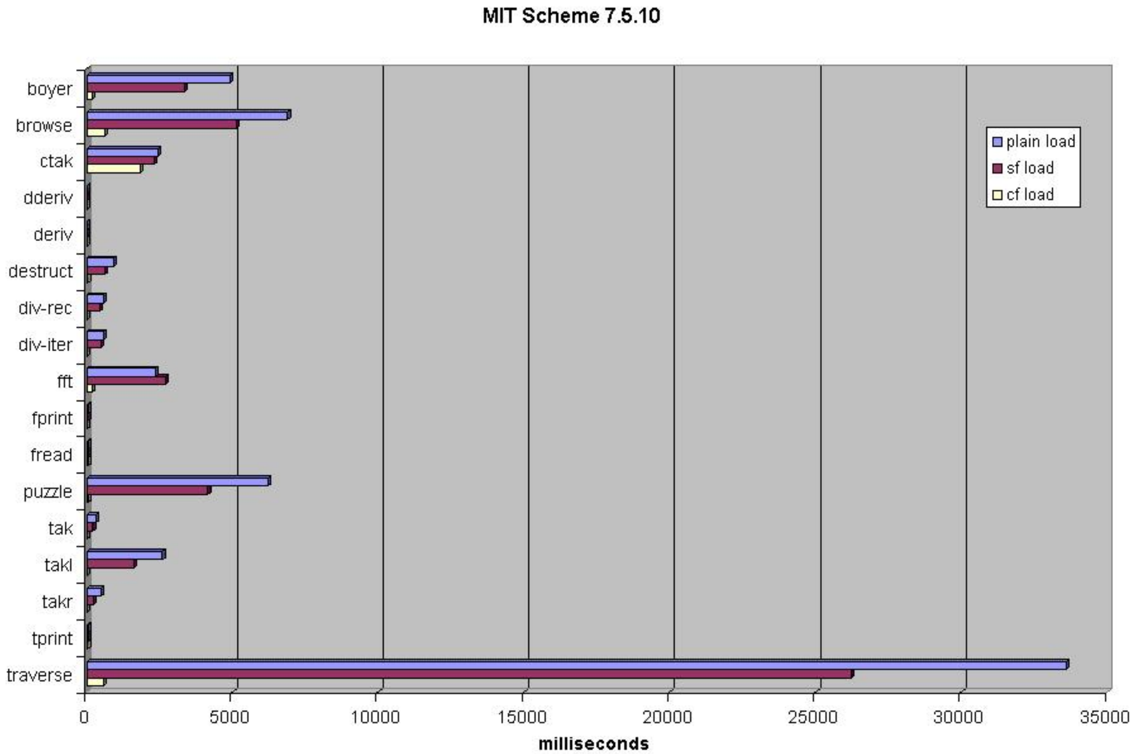


Even with `,bench`, Script does better on `browse` and `fft`. However, Scheme 48 does really clobber Script on `boyer`, `ctak`, and `puzzle`. For `traverse` things are closer until `,bench` is turned on, where Scheme 48 widens the gap, as in many of the other tests.

MIT Scheme

MIT Scheme is the product of the MIT Project on Mathematics and Computation. It provides a native-code compiler, the only such compiler in this survey.[44] [19]

MIT Scheme 7.5.10 was tested in three ways. First simple loading of Scheme code was tested. Second `sf` was used to do some syntax analysis and some optimzations. Third `cf` was used to compile the tests to run as native code.



The Script implementation's performance compares well when MIT Scheme simply loads or uses `sf`, with mixed results similar to Scheme 48. However `cf` leaves almost everything else in the dust. Kawa manages to come close to a tie on `browse`. One advantage MIT Scheme has is that the compilation is done as a separate step when Kawa is compiling the test each time they are run.

RScheme

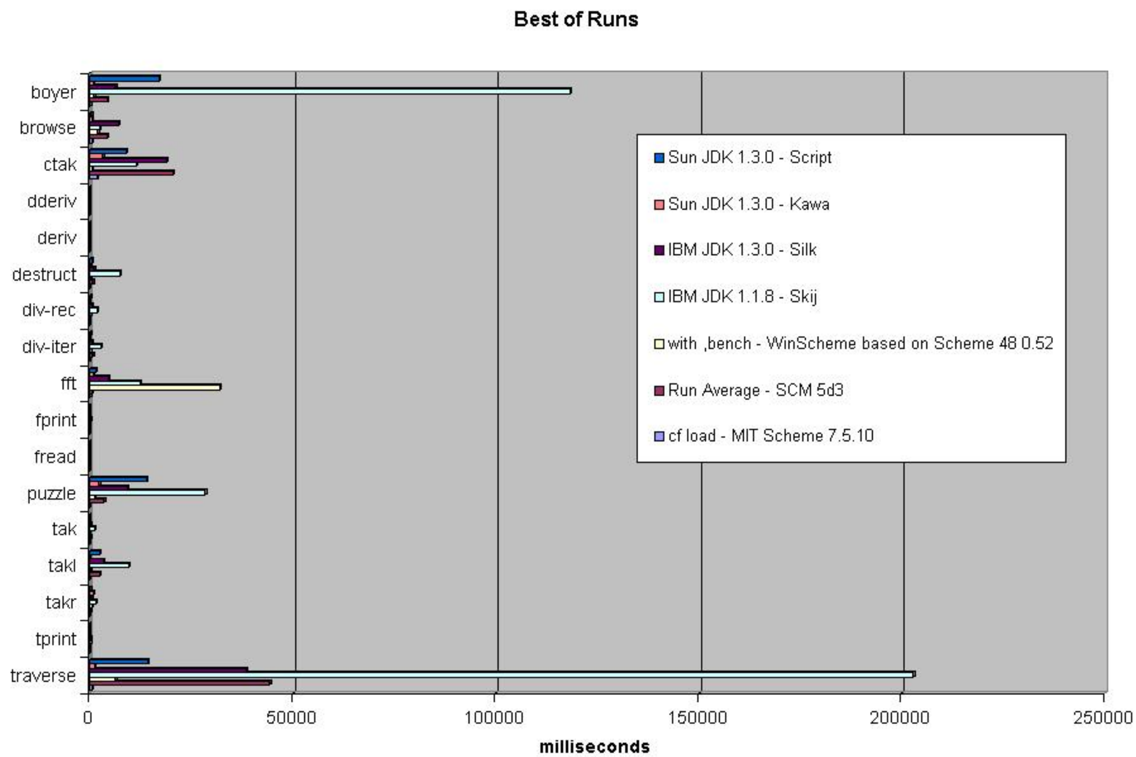
RScheme is a Scheme system from Donovan Kolbly. [31] Although it provides a full implementation of the language, unfortunately the system did not support the operating system of the test machine.

SIOD

SIOD, or Scheme in One Defun, is the product of George J. Carrette. [9] Unfortunately, SIOD is not really Scheme, lacking display and even write, so it was not able to run the benchmarks.

Overall

This section presents an overall comparison of the best performing runs of each Scheme implementation.



MIT Scheme clearly does the best overall, which is not much of a surprise given that it is the only system with a native code back-end. What is surprising is how close Kawa comes to matching it by generating only Java byte-codes, relying on the virtual machine's JIT compiler to produce native code. Slightly behind the leaders is Scheme 48. Scheme 48 has an surprising last place finish on the `fft` test, although on some tests it fairs well with the top contenders. Script and SCM fall in the middle of the pack, with no last place finishes. Silk comes in behind these two with one last place finish. Skij comes in last, not surprising given its s-expression interpretation.

9.2 Comparative Analysis with other Scheme-like Java systems

This section provides brief overviews of other Scheme-like Java systems. Some intend to provide a Scheme system but were not complete enough to run the Gabriel benchmarks. Some only claim to be similar to Scheme or Lisp but provide similar execution strategies and extensions to Scheme systems.

9.2.1 The scheme package

The scheme package is the product of Stéphane Hillion. [22] Version 1.1 was tested but `fft` failed to run. This problem was reported to the author but no response was received. As usual, `call-with-current-continuation` is limited to escape and error procedures. There are extensions for reflection, bit manipulation, asserts, and batch processing.

9.2.2 PS3I

PS^{3}I is a Scheme implementation from Christian Queinnec. It replaces the earlier Jaja system from which it borrowed only its reader. It provides a command line and servlet interface. Remarkably PS^{3}I supports full continuations although it is an s-expression interpreter reling heavily on Java reflection hurting its performance. Unfortunately in revision 1.18 many standard functions such as `atan`, `expt`, and even `write` were missing preventing the Gabriel benchmarks from running. PS^{3}I supports the mixed use of `Strings` and `StringBuffers` for Scheme `strings` and uses `Object[]s` for Scheme `vectors`. There are extensions for exceptions, threads, dynamic variables, and inherited thread locals. `worlds` provide first class environments.

9.2.3 LISC

LISC, also known as LIghtweight Scheme on Caffeine, was written by Scott G. Miller. [43] Version 1.2.3 was tested but `atan` and `expt` and other standard procedures were missing. LISC uses an s-expression based interpreter. It has extensions for first class environments and data triggers.

9.2.4 HotScheme

HotScheme is a project from Gene Callahan, Brian Clark, Rob Dodson, and Prasad Yalamanchi. [8] HotScheme provides a command line and `applet` environment. HotScheme contains no version information. The version tested was missing many standard features such as `call-with-current-continuation`, syntactical sugar for `define` and `lambda`, n-ary arguments to lambda, `integer` arithmetic, `atan`, and `expt`. It does have `load`, which uses URLs like the Script implementation. Similar to Script and Kawa named procedures, HotScheme allows a name and usage information to be associated with procedures.

9.2.5 MIT Scheme in Java

Arjuna Wijeyekoon provides something called MIT Scheme in Java. [66] It is not clear how it is related to MIT Scheme. It is simply available as an `Applet` from a web page without any other documentation. A broken `atan` and other problems prevented the Gabriel benchmarks from running.

9.2.6 PAT

PAT, the Performance Analysis Tool, by Joshua S. Allen is available from IBM alphaWorks. [2]. It is Scheme-like but does not pretend to be Scheme. It can run as an interactive application with a built-in help system. It offers extensions for reflect, dates, set operations, and statistics. It can serialize Java objects to and from XML.

9.2.7 LispkitLISP Compiler in Java

The LispkitLISP Compiler in Java was written by Chris Walton. [65]. It implements the SECD virtual machine in Java and uses a compiler to compile a simple Lisp subset to this virtual machine.

9.3 Comparative Analysis with other Java extension systems

When the Script implementation was started only a commercial Basic interpreter was available for Java. Now many languages have been ported to the Java environment. This section reviews many other language systems that are available for the Java platform. For more information, Robert Tolksdorf maintains a list of languages projects related to the Java platform. [60]

9.3.1 HotTea

HotTEA is a Basic interpreter from Michael G. Lehman of Cereus7. [38]. It compiles BASIC into a byte-code form which is then interpreted in Java. There are three different versions. URLGrey which is compact and can run as an `Applet`. Green extends URLGrey with compatibility with Microsoft Visual Basic for Applications and extensions for reflection and JavaBeans. [41] BRISK extends Green to be embeddable by Java applications authors.

9.3.2 Rhino

Rhino is a JavaScript interpreter from Mozilla. [50] It technically follows the ECMAScript standard but supports extensions to the language common to both Netscape Navigator and Microsoft Internet Explorer. [13] [40] The original releases from Mozilla were interpreted only but a Java byte-code compiler has been donated by Netscape as well.

9.3.3 Jacl

Jacl is a Tcl interpreter in Java originally by Sun Laboratories now maintained by Scriptics. [51] It uses reflection to interact with Java. SWANK provides a Tk toolkit implemented using the Java Swing toolkit. [26] It currently has some problems running in browsers and does not yet support the full Tcl language.

9.3.4 JPython

JPython is a Java implementation of the Python language. [49] It compiles Python to Java byte-codes either dynamically or statically. JPythons performance on the pystone benchmark can beat that of CPython on the same machine depending on the Java virtual machine. It allows JPython classes to extend Java classes as well as reflection and JavaBean support.

9.3.5 BeanShell

BeanShell is a scripting language from Pat Niemeyer. [6] BeanShell's syntax is very similar to Java itself. BeanShell's object model is distinct from Java's. BeanShell does not allow creating new subclasses of Java classes. Objects are closures like in Perl or JavaScript. It supports the JavaBean, from which it derives its name, as well as reflection. It can operate in a `Applet`, console, or RMI server environment. Even though it is simular to Java, it does attempt to compile to Java byte-codes. BeanShell is used as the Java source interpreter for JDE, the Java Development Environment for Emacs. [30]

9.3.6 DynamicJava

DynamicJava is a scripting language from Stéphane Hillion, also author of the scheme package. [23]. DynamicJava is similar in concept to BeanShell, but is completely source code compatible with Java. Because this is truly the case, DynamicJava allows subclassing of Java classes. Although DynamicJava does some byte-code generation to allow generating dynamic subclasses that invoke interpreted code, it does not provide a general Java byte-code compiler. DynamicJava extends the Java language by allowing statements outside of classes and methods, optional variable declarations, optional casting, package switching, classless methods, and `#` comments. It separates out the parser to allow other language front ends to be plugged in. One minor bug still remaining is that DynamicJava does not correctly intern string literals.

Chapter 10

Future Work

One area of future work is to bring the implementation closer to R^{5}RS compliance. The `Reader` should be enhanced to support the syntax for `vectors`. Internal `defines` should be easy to add with `Compiler` work to scan them out and replace them with a `letrec`. Similarly support for named `let` should be possible by enhancing `Let2Application`. Pseudoscheme-style analysis could then be performed to translate self tail calls into loops.

The `GlobalEnvironment` currently allows for only one environment. To implement R^{5}RS `eval` there must actually be several different environments: the `null-environment`, `scheme-report-environments`, and the `interaction-environment`. Language embedders would also like to have internal control over the environments. For example although multi-engine was desired to provide isolation, the `scheme-report-environment` could be shared reducing initialization cost. Searching a nested set of `GlobalEnvironments` should have little impact from a performance point of view since determining the right `GlobalEnvironment` happens at compile-time and not run-time. Note that this is not the same as first-class environments but is seen by the language embedder. Rhino provides this functionality and an implementation might choose to have every script called in a clean new environment.

With Java reflection, most library needs can be satisfied outside the implementation. Prior to JDK 1.3 reflection was focused on the dynamic invocation of Java code. In JDK 1.3, reflection was enhanced so that a class could be made to dynamically implement an interface. With this functionality, the Script implementation could use Scheme functions to implement Java interfaces. Since interfaces are commonly used for call-back APIs such as in UI toolkits, this would help further eliminate the need for Java coding to interface

Scheme to the Java class libraries.

Although reflection can be used to manipulate `Object[]`, as shown above, this is very expensive. It would probably be cheaper to update the `vector-*` primitives to handle both `Vectors` and `Object[]` similar to how `string-*` operations work with both `Strings` and `char[]`. Since the code currently checks the argument type to ensure that a `Vector` is passed adding a second case would not slow down the common `Vector` case although the second `Object[]` case would be a slower.

The current implementation relies on both Java class files and Scheme source files to be present. This packaging issue could be simplified if the system and utility Scheme code could be converted into constants in well known Java classes. The implementation could conditially load the system and utility code from Strings stored in Java classes if they are present, removing the run-time dependency on files, making everything class files.

Longer term the overall interpretation stategy could be rethought. One possability might be move to a Scheme-specific virtual machine on top of Java similar to Scheme 48 to remove the use of the Java stack, supporting general tail recursion and possible full continuations. Another might be to take the Kawa approach of generating Java byte-code.

Chapter 11

Conclusion

This section summarizes some of the lessons learned from this Scheme in Java implementation. It focuses on four different areas:

- Scheme-to-Java API
- Java-to-Scheme API
- Java performance
- Final thoughts

Many of the observations, especially regarding APIs, do not just apply to Scheme in Java. Specifically, the lessons could be applied when embedding other languages in Java or when embedding Scheme in other languages.

11.1 Scheme-to-Java API

The Scheme-to-Java API focuses on providing the standard Scheme library as well as access to application and user extensions. There four general ways to do this:

- Java registration of Java primitives
- Scheme registration of Java primitives
- Scheme implementation using Java reflection functions
- Scheme implementation using Scheme functions

The first case is unavoidable to some degree but unforunately makes it more difficult to seperate maintenance of the language system from the addition of primitives. The second case is a simple improvement on the first, allowing applications to add their own primitives without having to make changes to the underlying language system. The third case improves on the second by removing the need for any new Java programming at all but at the cost of the overhead of reflection. The fourth case is to simply avoid using Java to build things than can be built in Scheme itself, possibly sacrificing run-time performance for a simpler implementation.

Another important aspect of the Scheme-to-Java API is providing the right supporting APIs to the authors of Java primitives. The key here is to make the simple things simple and the complex things possible. Specifically, it should be easy to write new primitives with a fixed number of arguments that use standard classes as arguments. Layered on top of that, it should be possible to pass in application specific classes, handle n-ary argument functions, functions with defaulted arguments, etc.

11.2 Java-to-Scheme API

A well-designed Java-to-Scheme API has several aspects:

- the general architecture and its limitations
- the Java environments it supports
- the call API and the operations it exposes
- the general programming environment it supports

The general architecture limits how the embedded language can be used. A language system that is not safe for multi-thread could be useful for a REPL and even for scripting an event-driven user interface although would not provide scalable server-side scripting. A system that does not allow multiple interpreters per virtual machine is still generally useful but does prevent a complex application from partitioning and isolating its various uses of scripting. Similarly a system that does not run in an `Applet` environment prevents sophisticated tools with graphical user interfaces from being deployed through web browsers.

The Java environment affects the deployability of a system. Requiring only JDK 1.0 means that the system can work on even Netscape Navigator 3.x and Internet Explorer 3.x. JDK 1.1 means requiring the 4.x version of those browsers but adds internationalization and the ability to include a reflection API. JDK 1.2 provides builtin thread-local storage, the Swing UI toolkit, and new collection classes but limits the ability to run in most browsers. JDK 1.3 provides even more new APIs but is not yet widely available on all operating system platforms. For maximum flexibility it seems wise to keep the core part of the language system on the lowest version possible and then provide optional libraries to provide the newer APIs. JDK 1.1 seems like a reasonable lower bound because proper internationalization needs to be part of the core system and 4.x browsers are relatively standard.

The Java call needs to be well composable to meet the broadest application needs possible. Here again a mantra of making the simple things simple and the complex things possible applies. The API started out allowing an application to load a file, lookup a procedure, and call it with some arguments, each with builtin error handling. Later these operations were broken down into their component parts to make things more flexible. Loading a file was separated into reading from a stream into an s-expression, compiling an s-expression to a `Expression`, and evaluating an `Expression` to get an result, each exposing possible exceptional conditions to their callers. Looking up a procedure was broken down into getting or setting a global variable also throwing exceptions, this time possibly for undefined variables. Instead of just calling a procedure once, creating a new `Application Expression` each time, the `Application` can be reused and later evaluated like any other `Expression`. Even the simpler high-level APIs added options such as rethrowing of certain `RuntimeExceptions` and surpression of warnings.

Proper tools need to be provided by the programming environment to make both script and application authors successful. This may seem obvious but too often Scheme systems often seen to be written for the personal uses of their authors. Scheme's minimalist philosophy seems to lead to spartan environments as well. Simple source-level debugging needs to be provided to script authors so they can find their problems easily. Stack traces needed to be available to provide context in tracking down these problems. Application authors are often script authors as well, but in addition need help debugging their Java primitives as well. Both script and application authors need to have the particular details of the implementation hidden from them as much as possible so they can focus on what is wrong in their part of the system.

These areas are not all independant of course. Support for providing file and line number information is available only because the system is architected to provide it and only accessible because of the proper exception API. The potential future feature of nested global environments will change how the compiler works as well as call API and even possible the development environment.

11.3 Java Performance Lessons

This implementation started as a project to learn about Java. The most important lessons learned were about performance. The general lessons learned were:

- Thou shall not synchronize
- Thou shall not allocate
- Thou shall not abuse exceptions
- Thou shall not forsake buffering
- Thou shall not forsake arrays
- Thou shall honor pointer equality

When discussing these lessons, it is important to realize although some of the details are Java specific, the concepts apply to other systems as well.

11.3.1 Thou shall not synchronize

The most straightforward reason to avoid synchronization is that it does not come for free. This is compounded by the fact that the expected performance is non-intuitive on virtual machines that optimize for memory usage instead of scalability. The number of CPUs accessing a given monitor can increase the cost of synchronization as well, limiting scalability.

The easist way to avoid the cost of synchronization is to avoid implicitly synchronized class such as the standard `java.lang.StringBuffer`, `java.util.Vector`, and `java.util.Hashtable` either by using third-party alternatives or the new JDK 1.2 collection classes. Unfortunately there is still no standard alternative to `StringBuffer` which is particular problematic given its implicit use realted to the `String +` operator.

Even when synchronization is necessary, it is often better to perform explicit locking with the unsynchronized classes rather than gain a false sense of security using the implicitly synchronized classes, as demonstrated with the JDK 1.0 `String.intern` bug. Even when threads need to share a data-structure, it need not be synchronized, as shown by `GrowOnlyHashtable`.

11.3.2 Thou shall not allocate

Allocating memory is expensive not only at time of allocation but also because of the later cost of garbage collection. Depending on the virtual machine, memory allocation can imply synchronization on a single underlying heap. Garbage collection has its inherent costs but scalability is also an issue. On a multi-processor machine simple collectors may stall otherwise ready CPUs while collection proceeds.

As in the physical world, the mantra "reduce, reuse, recycle" can serve as a guide to reduce unnecessary memory and other resource allocation. To reduce allocation, avoid allocating intermediate results. For example use a mutable, but unsynchronized, `StringBuffer` and convert to a `String` at the end of an calculation, rather than using `Strings` throughout. To reuse resouces, use pools or caches such as `getInteger`. Pooling is key for other expensive system resources such as threads and database connections.

11.3.3 Thou shall not abuse exceptions

Exceptions should be used for exceptional conditions, not for normal control flow. Although this is primarily a performance consideration for `jdb` at development-time, not for `java` at development time, it can seriously impact developer productivity. In addition to the time lost when running in `jdb`, unnecessary `RuntimeExceptions` can make it hard to track down real problems. For example JDK 1.1's `java.text.*` classes would often throw `NullPointerExceptions` and `IndexOutOfBoundsException` in their normal course of operation of attempting to parse different formats. Unfortunately this meant that telling a debugger to stop on `NullPointerException` would encounter a lot of false problems. Some other bad examples are Weblogic and `javax.mail` which do not use `File.exists` to see if a file exists, but instead catch `IOException`.

11.3.4 Thou shall not forsake buffering

Reading and writing characters one at a time without buffering is painfully slow in any language. Java internationalization adds a new twist. Even if a stream of bytes is buffered, the one at a time conversion from characters to bytes and bytes to characters is just as bad.

For example when reading characters from a file, it is very important to use a pipeline of `BufferedReader`, `InputStreamReader`, and `FileInputStream`. The `BufferedReader` batches requests for characters to the `InputStreamReader` which in turn batches requests for bytes to the underlying `FileInputStream`.

The dangerous alternative is to use a pipeline of `InputStreamReader`, `BufferedInputStream`, and `FileInputStream`. Although the `BufferedInputStream` batches requests for bytes to the `FileInputStream`, the `InputStreamReader` will only convert a character at a time.

Another example of the advantage of avoiding one-at-a-time operations is using `System.arraycopy`. In addition to having a native implementation, `System.arraycopy` is superior to a Java copy loop because it performs bounds checks once on each array instead of once for each access.

11.3.5 Thou shall not forsake arrays

`Vectors` are very heavily used in Java. Their encapsulation of sizing is very useful. Unfortunately this encapsulation comes with the cost of method-call overhead to access elements and length information. In addition, APIs trafficking in `Vectors` do not provide compile-time type safety.

Object arrays provide an alternative. Unfortunately in exchange for type safety and improved access speed comes the pains of manual sizing. A `Vector`-like class that provides automatic resizing and type safety through subclassing while exposing the underlying array for more efficient and type free access is a good compromise.

11.3.6 Thou shall honor pointer equality

A small but important point is to take advantage of pointer equality whenever possible. In this system that meant avoiding `String.equals` by interning `Symbols`. The implementation assumed that `Boolean.FALSE` was the only `false Boolean` value. This is a general safe assumption, although there is no way to prevent someone from using `new Boolean(false)`, leaving one to wonder why the constructor is even `public`. Finally, another way to take advantage of pointer equality is to use hashtables that rely on `==` equality instead of `equals`.

11.4 Final Thoughts

An important lesson learned was to minimize special cases and keep things simple. When special cases and complexity are added, they should have a clear purpose and goal. The simplification and cleanup in the second pass, especially of the `Expression` and `Object` mixup, revealed this. By the end of the second pass a simple modular implementation allowing for more iterative change in later passes.

In the end, this Scheme in Java implementation served its purpose by quickly providing a scripting extension language. However over time as other scripting languages were made available in Java, the unfamiliarity of the Scheme language to the average system implementor led the embedding application to seek out other solutions. In the end the application chose to support extensions in Scheme, JavaScript, and Java.

Chapter 12

Acknowledgments

Thanks to my wife Jennifer for all her support and patience in getting my thesis completed. Thanks to my mom for nagging me to get my masters when most people no longer thought it was that important. Thanks to my dad for introducing me to Lisp at the impressionable age of twelve. It makes up for introducing me to Basic at the age of six which Dijkstra says should have rendered me incapable of properly learning to program. Of course maybe he was right. Thanks to my sister Rachael for having nothing to do with computers but hopefully still thinking I'm a cool brother, even if it is because I bribe her.

Thanks to Olin Shivers and Norman Adams for indoctrinating an MIT Scheme student with a Yale Scheme point of view and generally showing me the ways of the force. Thanks to Michael Blair, better known as Ziggy, for my first introduction to Scheme as my TA in 6.001. Thanks to Franklyn Turbak to helping me get it right in 6.821. Thanks to my friends Jason Wilson, David LaMacchia, Brian Zuzga, and Natalya Cohen, the Switzerland Summer of 1992 UROPs, that made all my course VI classes bearable. Thanks to Arthur Gleckler, Philip Greenspun, Elmer Hung, Brian LaMacchia, and Rajeev Surati, the Swiss elders for their help, advice and friendship.

Thanks to Lucille Glassman for reviewing this document and providing primary and secondary DNS name service. Thanks to Dmitri Schoeman for helping me squeeze out the last ounce of performance out of the Java run-time as well as some last minute encouragement, reviewing, and Krispy Kreme doughnuts. Thanks to Stephanie Shaw and Simanta Chakraborty for providing a place to stay with high speed Internet access while at the 'tute. See Stephanie, I did finish my thesis before you. And last by certainly not least, thanks to

Anne Hunter for getting me through the MIT bureaucracy a few years late, but hey, better late than never.

Appendix A

Benchmark Results

All tests were performed on a Dell Dimension T800r system with an 800MHz Intel Pentium III processor with 512MB of RAM running Windows 2000 Professional Service Pack 1.

A.1 Script

A.1.1 Sun JDK 1.3.0

#	boyer	browse	ctak	dderiv	deriv	destruct	div-rec	div-iter	fft	fprint	fread	puzzle	tak	takl	takr	tprint	traverse
1	17244	470	9194	10	0	731	410	331	1692	60	20	14071	110	2573	161	40	14310
2	17205	471	9153	10	10	741	401	330	1672	71	10	14020	110	2574	150	40	14381
3	17275	481	9163	10	10	741	411	330	1703	60	10	14030	120	2554	160	40	14311
4	17205	481	9163	0	0	741	421	331	1682	60	10	14020	120	2574	150	40	14321
5	17174	460	9163	10	0	742	400	341	1682	60	20	14040	121	2553	160	40	14311

A.1.2 Sun JDK 1.2.2

#	boyer	browse	ctak	dderiv	deriv	destruct	div-rec	div-iter	fft	fprint	fread	puzzle	tak	takl	takr	tprint	traverse
1	30494	701	6149	10	20	1022	531	540	3596	130	50	26648	170	4697	291	90	19398
2	30424	701	6109	40	10	1031	531	541	3595	120	40	26869	170	4737	290	91	19267
3	30444	681	6108	41	20	1031	521	541	3605	120	40	26748	171	4787	240	120	19138
4	30423	691	6149	40	10	1052	560	571	3505	120	50	26669	180	4727	300	90	19278
5	30404	691	6119	30	10	1052	551	530	3606	120	50	26758	181	4716	301	100	19248

A.1.3 Sun JDK 1.1.8

#	boyer	browse	ctak	dderiv	deriv	destruct	div-rec	div-iter	fft	fprint	fread	puzzle	tak	takl	takr	tprint	traverse
1	38675	731	15913	10	10	1092	651	571	4176	160	20	37965	170	5908	221	190	20549
2	38675	731	15913	10	0	1092	651	561	4166	170	20	37914	171	5908	240	191	20549
3	38666	741	15923	10	10	1082	651	560	4156	171	30	37864	160	5909	220	201	20529
4	38676	751	15933	10	0	1082	650	561	4166	160	30	37864	160	5919	220	200	20540
5	38676	741	15913	10	0	1091	651	571	4166	170	30	37925	160	5919	230	210	20520

A.1.4 IBM JDK 1.3.0

#	boyer	browse	ctak	dderiv	deriv	destruct	div-rec	div-iter	fft	fprint	fread	puzzle	tak	takl	takr	tprint	traverse
1	40278	1092	11817	10	0	2012	731	1142	3946	190	10	28731	211	5357	251	30	41279
2	40619	1072	11687	20	0	2003	711	1111	3866	190	10	28451	210	5158	270	30	41300
3	36953	1062	11727	10	0	2053	671	1061	4236	201	10	29472	230	5258	270	40	40369
4	36933	1102	11867	20	0	1993	631	1091	4377	190	10	29623	230	5268	270	40	40608
5	36802	1062	13078	30	10	2053	651	1092	4266	130	20	29703	230	5278	280	60	41130

A.1.5 IBM JDK 1.1.8

#	boyer	browse	ctak	dderiv	deriv	destruct	div-rec	div-iter	fft	fprint	fread	puzzle	tak	takl	takr	tprint	traverse
1	46537	1061	11337	10	10	2373	801	1282	4527	80	10	33568	291	6158	331	60	48239
2	46406	1041	11436	10	11	2373	781	1282	4506	101	10	33828	281	6188	351	140	47749
3	46187	1051	11457	10	10	2393	791	1262	4537	90	10	33528	270	6179	351	50	47799
4	46637	1062	12197	10	10	2374	811	1272	4486	90	10	33529	270	6149	340	131	48159
5	46357	1051	11347	10	10	2363	791	1282	4537	100	20	33498	280	6239	331	60	47188

A.1.6 Microsoft SDK for Java 3.1

#	boyer	browse	ctak	dderiv	deriv	destruct	div-rec	div-iter	fft	fprint	fread	puzzle	tak	takl	takr	tprint	traverse
1	20389	501	10615	0	10	891	381	380	2264	40	90	17195	150	2944	190	40	16334
2	20310	510	10586	0	10	881	391	390	2243	40	91	17224	151	2924	190	40	16324
3	20329	491	10595	10	0	892	380	401	2253	40	90	17185	150	2944	191	40	16333
4	20309	501	10575	10	0	911	391	390	2254	40	90	17165	140	2944	200	40	16304
5	20319	500	10596	10	10	881	390	391	2233	40	90	17355	151	2944	190	40	16314

A.2 Kawa

A.2.1 Sun JDK 1.3.0

#	boyer	browse	ctak	dderiv	deriv	destruct	div-rec	div-iter	fft	fprint	fread	puzzle	tak	takl	takr	tprint	traverse
1	851	721	3285	80	40	271	70	70	1081	70	40	2334	30	70	851	40	1392
2	861	701	3325	80	30	271	70	70	1051	71	40	2343	30	70	821	30	1332
3	861	741	3295	80	40	271	70	70	1071	71	30	2343	30	70	821	40	1372
4	871	711	3305	80	30	271	70	70	1051	71	40	2353	30	70	821	40	1342
5	871	721	3285	80	40	271	70	70	1081	71	40	2333	30	70	831	40	1382

A.2.2 Sun JDK 1.2.2

#	boyer	browse	ctak	dderiv	deriv	destruct	div-rec	div-iter	fft	fprint	fread	puzzle	tak	takl	takr	tprint	traverse
1	1512	1232	1813	90	50	190	141	140	1111	80	81	1942	40	71	901	40	1963
2	1542	1212	1792	100	80	131	140	130	1272	90	70	1863	40	70	921	30	1943
3	1572	1182	1813	100	50	160	130	140	1112	80	80	1693	30	70	871	40	1993
4	1562	1222	1763	100	50	170	140	130	1112	80	80	1733	30	70	891	40	1983
5	1512	1222	1813	100	50	180	140	130	1112	80	80	1913	30	70	901	41	1952

A.2.3 Sun JDK 1.1.8

#	boyer	browse	ctak	dderiv	deriv	destruct	div-rec	div-iter	fft	fprint	fread	puzzle	tak	takl	takr	tprint	traverse
1	1893	1402	5889	90	50	180	150	131	1341	61	40	2213	70	70	921	30	2053
2	1893	1432	5879	100	40	190	140	141	1362	50	50	2203	70	80	911	40	2053
3	1893	1392	5859	100	50	180	150	131	1361	61	50	2203	70	70	911	40	2063
4	1883	1402	5839	100	50	180	150	130	1332	60	51	2223	70	70	921	40	2063
5	1873	1392	5849	100	50	180	150	140	1342	60	41	2193	70	70	911	50	2043

A.2.4 IBM JDK 1.3.0

#	boyer	browse	ctak	dderiv	deriv	destruct	div-rec	div-iter	fft	fprint	fread	puzzle	tak	takl	takr	tprint	traverse
1	941	1152	3745	120	80	351	90	90	681	110	40	1042	30	70	1452	50	1652
2	931	1152	3745	111	80	350	90	91	691	120	40	1021	30	70	1452	51	1632
3	931	1162	3745	110	80	351	100	80	681	120	40	1042	30	70	1462	60	1653
4	962	1171	3736	120	70	381	90	90	681	120	40	1031	31	70	1452	60	1662
5	931	1152	3765	111	80	350	90	80	681	121	30	1051	30	70	1452	50	1653

A.2.5 IBM JDK 1.1.8

#	boyer	browse	ctak	dderiv	deriv	destruct	div-rec	div-iter	fft	fprint	fread	puzzle	tak	takl	takr	tprint	traverse
1	961	1011	3906	140	60	331	90	110	962	60	30	1262	30	70	1312	90	1532
2	971	1002	3925	140	80	331	90	110	992	70	30	1282	30	80	1322	40	1542
3	951	992	3935	140	71	320	90	110	962	70	30	1282	30	70	1312	40	1542
4	971	972	3785	150	70	321	90	110	971	71	30	1221	30	61	1311	61	1582
5	961	971	3796	150	60	351	90	100	941	70	30	1242	30	70	1292	40	1573

A.2.6 Microsoft SDK for Java 3.1

#	boyer	browse	ctak	dderiv	deriv	destruct	div-rec	div-iter	fft	fprint	fread	puzzle	tak	takl	takr	tprint	traverse
1	931	741	7621	50	30	120	80	81	761	30	40	1161	30	81	600	20	1753
2	932	751	7721	50	30	130	80	71	761	30	40	1141	40	81	610	20	1763
3	931	761	7691	50	30	130	71	70	761	30	40	1152	30	80	601	20	1762
4	931	761	7681	40	30	130	80	71	751	40	30	1151	30	71	610	20	1783
5	941	761	7861	50	30	121	90	70	761	30	40	1152	30	70	591	30	1752

A.3 SILK

A.3.1 Sun JDK 1.3.0

#	boyer	browse	ctak	dderiv	deriv	destruct	div-rec	div-iter	fft	fprint	fread	puzzle	tak	takl	takr	tprint	traverse
1	9674	11887	11186	10	10	1762	1012	1102	6098	80	10	14281	641	4877	711	40	61308
2	9614	11827	11216	10	0	1763	1021	1102	6099	80	20	14210	641	4867	711	40	61288
3	9613	11827	11217	10	0	1762	1012	1101	6099	80	10	14221	630	4857	711	41	61398
4	9634	11817	11186	10	10	1753	1011	1112	6118	90	11	14210	631	4857	721	40	61258
5	9654	11827	11196	10	0	1763	1001	1112	6099	80	10	14250	641	4857	701	60	61248

A.3.2 Sun JDK 1.2.2

#	boyer	browse	ctak	dderiv	deriv	destruct	div-rec	div-iter	fft	fprint	fread	puzzle	tak	takl	takr	tprint	traverse
1	15902	13139	4917	10	0	2023	1142	1262	7320	121	60	18116	711	5528	891	60	93935
2	15893	13129	4927	20	10	2033	1172	1232	7330	120	50	18097	701	5497	862	60	93845
3	15913	13139	4907	20	10	2033	1161	1242	7341	120	50	18126	711	5538	892	60	93915
4	16003	13259	4947	20	10	2073	1152	1272	7370	80	60	18016	751	5438	882	60	94916
5	15973	13159	4917	20	10	2023	1152	1262	7320	121	50	18126	711	5528	911	60	93945

A.3.3 Sun JDK 1.1.8

#	boyer	browse	ctak	dderiv	deriv	destruct	div-rec	div-iter	fft	fprint	fread	puzzle	tak	takl	takr	tprint	traverse
1	23975	16233	15352	10	10	2514	1502	1522	9875	130	30	22362	931	6710	1041	81	126752
2	23965	16243	15352	10	0	2504	1502	1522	9874	120	30	22383	901	6730	1041	80	126772
3	23974	16234	15342	10	20	2483	1502	1543	9874	120	30	22382	912	6719	1042	80	126722
4	23975	16233	15342	20	0	2504	1512	1532	9904	121	20	22342	931	6720	1041	80	126773
5	23984	16234	15332	10	10	2493	1503	1532	9904	120	20	22332	922	6709	1042	80	126822

A.3.4 IBM JDK 1.3.0

#	boyer	browse	ctak	dderiv	deriv	destruct	div-rec	div-iter	fft	fprint	fread	puzzle	tak	takl	takr	tprint	traverse
1	6449	7070	19007	0	10	1122	711	621	4717	70	20	9493	391	3395	471	70	38896
2	6379	7101	18987	10	10	1112	691	621	4836	101	10	9463	391	3335	480	50	39177
3	6380	7100	18987	0	0	1122	691	631	4827	100	20	9453	391	3345	470	50	37935
4	6419	7040	18977	0	10	1142	681	621	4717	80	10	9383	381	3305	460	50	38936
5	6399	7110	19068	10	10	1142	701	640	4867	101	10	9503	381	3395	480	50	38255

A.3.5 IBM JDK 1.1.8

#	boyer	browse	ctak	dderiv	deriv	destruct	div-rec	div-iter	fft	fprint	fread	puzzle	tak	takl	takr	tprint	traverse
1	8623	9864	16844	0	0	1492	1022	881	5468	100	10	11767	511	4746	621	30	53007
2	8643	9864	16814	10	10	1502	1012	891	5518	80	20	11807	511	4777	610	51	53246
3	8572	9894	16825	10	10	1492	1021	901	5498	80	10	11796	501	4767	601	40	54038
4	8553	9874	16814	0	10	1522	1032	891	5518	80	10	11677	501	4736	611	40	52746
5	8652	9975	16824	0	10	1482	1031	902	5538	80	10	11887	511	4746	631	40	52997

A.3.6 Microsoft SDK for Java 3.1

#	boyer	browse	ctak	dderiv	deriv	destruct	div-rec	div-iter	fft	fprint	fread	puzzle	tak	takl	takr	tprint	traverse
1	12038	12297	16284	10	0	1842	1272	952	6309	60	20	20770	681	4696	862	60	59455
2	12047	12268	16243	10	10	1863	1272	941	6309	60	21	20769	681	4697	871	50	59476
3	12047	12288	16243	10	0	1863	1282	941	6319	50	20	20750	671	4707	871	50	59476
4	12097	12298	16263	10	10	1853	1262	951	6339	70	20	20740	671	4707	881	50	59496
5	12077	12288	16253	10	10	1863	1272	941	6329	50	20	20740	691	4697	881	50	59466

A.4 Skij

A.4.1 Sun JDK 1.3.0

#	boyer	browse	ctak	dderiv	deriv	destruct	div-rec	div-iter	fft	fprint	fread	puzzle	tak	takl	takr	tprint	traverse
1	133292	3064	10735	31	20	6499	1983	2603	21571	390	20	34440	1131	9414	1312	330	380758
2	131219	3054	10836	20	20	6550	1972	2634	21621	401	10	34479	1112	9424	1311	341	381328
3	131149	3064	10816	30	20	6529	2003	2614	21661	420	20	34470	1122	9423	1312	321	381338
4	131158	3085	10795	20	20	6560	1993	2633	21672	420	20	34470	1101	9444	1312	330	381479
5	131208	3055	10825	30	20	6530	1983	2613	21691	401	40	34450	1101	9414	1332	320	381519

A.4.2 Sun JDK 1.2.2

#	boyer	browse	ctak	dderiv	deriv	destruct	div-rec	div-iter	fft	fprint	fread	puzzle	tak	takl	takr	tprint	traverse
1	165047	2574	7300	31	20	6539	2413	2915	15272	330	90	29302	1212	9844	1583	390	268606
2	165437	2594	7271	80	20	6790	2343	2944	15382	371	50	29632	1192	9864	1583	390	270159
3	164787	2604	7570	30	21	6709	2364	2944	15472	351	50	30023	1212	9924	1632	321	271630
4	164597	2574	7481	30	10	6669	2293	2985	15232	400	40	30044	1171	9794	1492	401	268196
5	164406	2614	7361	30	80	6719	2424	2964	15502	391	50	29883	1232	10134	1603	380	270369

A.4.3 Sun JDK 1.1.8

#	boyer	browse	ctak	dderiv	deriv	destruct	div-rec	div-iter	fft	fprint	fread	puzzle	tak	takl	takr	tprint	traverse
1	239755	2834	11336	20	20	7581	2714	3295	16473	381	30	32817	1392	11296	1713	440	316415
2	238593	2824	11236	20	20	7571	2694	3285	16484	390	30	32787	1392	11207	1722	421	316495
3	238613	2834	11237	20	20	7571	2723	3305	16464	380	30	32837	1382	11186	1712	421	316585
4	238473	2814	11256	30	20	7571	2694	3294	16464	381	30	32847	1382	11186	1712	431	316595
5	238583	2834	11226	20	20	7561	2704	3295	16504	380	30	32837	1392	11176	1713	421	316695

A.4.4 IBM JDK 1.3.0

#	boyer	browse	ctak	dderiv	deriv	destruct	div-rec	div-iter	fft	fprint	fread	puzzle	tak	takl	takr	tprint	traverse
1	120874	2764	10565	20	20	7150	1773	2734	12187	411	10	28120	1252	9995	1532	400	202782
2	119923	2754	10475	20	20	7140	1763	2744	12217	411	20	28150	1252	10025	1502	400	202231
3	133411	3065	10725	20	20	8042	1803	3084	13920	481	10	32486	1392	11377	1662	401	229069
4	120363	2744	10725	30	20	7110	1773	2754	12187	451	10	27710	1212	9854	1432	380	203253
5	132160	3035	10745	30	20	7982	1802	3065	13920	420	80	32187	1331	11116	1593	400	233797

A.4.5 IBM JDK 1.1.8

#	boyer	browse	ctak	dderiv	deriv	destruct	div-rec	div-iter	fft	fprint	fread	puzzle	tak	takl	takr	tprint	traverse
1	118501	2513	11597	30	20	7421	1883	2794	12518	360	30	28511	1262	9724	1552	341	202631
2	118140	2543	11537	30	20	7371	1882	2824	12568	351	20	28571	1262	9604	1552	350	204645
3	118090	2534	11486	30	20	7351	1893	2764	12488	360	20	28511	1252	9654	1532	351	201900
4	118180	2513	11577	30	20	7371	1902	2764	12478	361	20	28270	1262	9524	1512	351	201740
5	118040	2534	11566	31	20	7360	1893	2774	12588	350	20	28642	1261	9744	1522	350	203323

A.4.6 Microsoft SDK for Java 3.1

#	boyer	browse	ctak	dderiv	deriv	destruct	div-rec	div-iter	fft	fprint	fread	puzzle	tak	takl	takr	tprint	traverse
1	296397	5297	20460	60	50	14961	7061	7280	18917	731	30	55771	2834	28571	3445	741	399084
2	296456	5328	20520	60	50	14981	7030	7281	18897	731	30	55810	2814	28531	3465	751	398744
3	296497	5297	20420	60	50	14991	7020	7281	18877	771	30	55790	2814	28541	3486	741	398793
4	296387	5297	20440	60	50	14961	7050	7281	18917	751	30	55881	2824	28551	3455	751	398913
5	296486	5308	20439	60	60	14952	7030	7270	18918	731	30	55760	2834	28561	3455	731	400706

A.5 WinScheme based on Scheme 48 0.52

A.5.1 without ,bench

#	boyer	browse	ctak	dderiv	deriv	destruct	div-rec	div-iter	fft	fprint	fread	puzzle	tak	takl	takr	tprint	traverse
1	2013	3445	691	60	50	511	240	250	32297	90	70	2353	121	1151	1032	130	11527
2	2043	3625	831	80	60	541	290	351	33528	90	80	2354	120	1152	991	210	11367
3	2043	3655	821	80	70	541	260	351	33508	80	70	2354	120	1162	991	200	11407
4	2043	3625	821	80	80	551	260	351	33518	90	80	2354	130	1152	991	200	11377
5	2033	3635	831	70	70	541	271	340	33598	90	71	2343	120	1162	971	230	11377

A.5.2 with ,bench

#	boyer	browse	ctak	dderiv	deriv	destruct	div-rec	div-iter	fft	fprint	fread	puzzle	tak	takl	takr	tprint	traverse
1	1081	2023	641	60	40	271	220	150	31636	80	120	1202	130	431	721	120	6449
2	1162	2073	731	70	40	280	221	150	32146	140	60	1272	60	431	681	120	6529
3	1091	2063	731	80	40	341	160	220	32097	70	70	1262	60	430	691	201	6389
4	1091	2073	731	60	51	340	160	221	32116	70	60	1282	60	441	670	201	6379
5	1082	2083	721	60	40	350	150	221	32106	70	70	1262	70	431	681	200	6329

A.6 SCM 5d3

#	boyer	browse	ctak	dderiv	deriv	destruct	div-rec	div-iter	fft	fprint	fread	puzzle	tak	takl	takr	tprint	traverse
1	5000	6000	22000	0	0	1000	0	1000	1000	0	0	4000	0	3000	1000	0	51000
2	4000	4000	21000	0	0	0	0	1000	1000	0	0	3000	0	3000	0	0	43000
3	4000	4000	20000	0	0	1000	0	1000	0	0	0	4000	0	3000	0	0	42000
4	5000	4000	20000	0	0	1000	0	1000	0	0	0	4000	0	2000	1000	0	43000
5	4000	4000	20000	0	0	1000	0	0	1000	0	0	3000	1000	2000	0	0	43000

A.7 MIT Scheme 7.5.10

A.7.1 plain load

#	boyer	browse	ctak	dderiv	deriv	destruct	div-rec	div-iter	fft	fprint	fread	puzzle	tak	takl	takr	tprint	traverse
1	5338	6880	2383	10	10	932	580	541	2334	50	110	6209	290	2574	480	61	33558
2	5077	6900	2403	10	10	942	581	681	2333	50	50	6279	301	2563	491	80	33689
3	4716	6880	2494	20	10	921	591	551	2433	40	50	6229	371	2574	490	60	33529
4	4707	6890	2483	20	10	922	590	541	2344	130	50	6189	300	2644	481	70	33638
5	4727	6900	2494	10	0	931	581	621	2343	40	50	6209	361	2604	480	60	33629

A.7.2 sf load

#	boyer	browse	ctak	dderiv	deriv	destruct	div-rec	div-iter	fft	fprint	fread	puzzle	tak	takl	takr	tprint	traverse
1	3415	5147	2274	0	0	600	450	471	2714	40	40	4146	230	1602	231	50	26218
2	3415	5138	2333	10	0	611	380	471	2714	30	50	4216	160	1603	300	50	26198
3	3335	5147	2333	0	10	601	451	480	2694	30	40	4156	161	1652	160	50	26188
4	3325	5168	2353	0	0	611	461	470	2714	40	40	4136	241	1592	170	130	26208
5	3335	5137	2353	0	0	621	451	481	2714	30	50	4156	240	1592	231	50	26247

A.7.3 cf load

#	boyer	browse	ctak	dderiv	deriv	destruct	div-rec	div-iter	fft	fprint	fread	puzzle	tak	takl	takr	tprint	traverse
1	210	761	1883	10	0	10	10	0	200	20	50	70	10	10	10	40	551
2	200	551	1833	0	0	10	90	10	190	20	40	60	10	10	10	40	581
3	191	560	1823	0	0	80	10	10	200	20	50	71	10	10	10	40	560
4	121	620	1843	0	0	10	10	70	130	20	121	70	10	10	10	40	551
5	201	550	1833	10	0	80	10	10	200	20	51	70	0	10	20	30	561

Bibliography

[1] Hal Abelson and Gerald Jay Sussman with Julie Sussman, *Structure and Interpretation of Computer Programs*, MIT Press and McGraw-Hill, (1985, Second edition 1996) `http://mitpress.mit.edu/sicp/`

[2] Joshua S. Allen, *Performance Analysis Tool*, `http://www.alphaWorks.ibm.com/tech/pat`

[3] Ken Anderson, Tim Hickey, and Peter Norvig, *SILK - A Java-based dialect of Scheme*, `http://www.cs.brandeis.edu/silk/silkweb/index.html`

[4] Ken Anderson, Tim Hickey, and Peter Norvig, *SILK - a playful blend of Scheme and Java*, `http://www.cs.brandeis.edu/$\sim$tim/Papers/scheme2000.html`

[5] BEA Systems, *WebLogic Server Performance Tuning Guide*, `http://www.weblogic.com/docs51/admindocs/tuning.html`

[6] Pat Niemeyer *BeanShell: Lightweight Scripting for Java*, `http://www.beanshell.org`

[7] Per Bothner, *Kawa, the Java-based Scheme system*, `http://www.cygnus.com/$\sim$bothner/kawa.html`

[8] Gene Callahan, Brian Clark, Rob Dodson, and Prasad Yalamanchi, *HotScheme*, `http://www.stgtech.com/HotScheme/`

[9] George J. Carrette, *SIOD: Scheme in One Defun*, `http://people.delphi.com/gjc/siod.html`

[10] Will Clinger, *Gabriel Benchmarks in Scheme*, `http://www.cs.indiana.edu/l/www/pub/scheme/gabriel-scheme.tar.Z`

[11] ECMA TC39/TG2, *C# Programming Language*, http://msdn.microsoft.com/net/ecma/

[12] ECMA TC39/TG3, *Common Language Infrastructure*, http://msdn.microsoft.com/net/ecma/ http://www.ecma.ch/ecma1/TOPICS/ECMA\%20CLI\%20Presentation.ppt

[13] ECMA, *Standard ECMA-262 ECMAScript: A general purpose,cross-platform programming language* (June 1997). http://www.ecma.ch/stand/ecma-262.htm

[14] Marc Feeley, James S. Miller, Guillermo J. Rozas, Jason A. Wilson, *Compiling Higher-Order Languages into Fully Tail-Recursive Portable C*, (18 March 1994).

[15] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns*, Addison-Wesley (15 January 1995). http://www.awl.com/product/0,2627,0201633612,00.html

[16] James Gosling, Bill Joy, Guy Steele, *The Java Language Specification*, Addison-Wesley (September 1996). http://java.sun.com/docs/books/jls/html/index.html

[17] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, *The Java Language Specification, Second Edition*, Addison-Wesley (5 June 2000). http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html

[18] Jim Gray and Andreas Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann (September 1992) http://www.mkp.com/books_catalog/catalog.asp?ISBN=1-55860-190-2

[19] Chris Hanson et al., *MIT Scheme Reference Manual*, MIT Artificial Intelligence Laboratory Technical Report 1281 (January 1991). http://www.swiss.ai.mit.edu/projects/scheme/documentation/scheme_toc.html

[20] Hewlett-Packard, *com.hp.io.Poll*, http://www.unixsolutions.hp.com/products/java/sdk12204rnotes.html

[21] Tim Hickney, *JScheme*, http://tigereye.cs.brandeis.edu/Applets/Jscheme.html

[22] Stéphane Hillion, *The scheme package*, `http://www-sop.inria.fr/koala/shillion/sp/`

[23] Stéphane Hillion, *DynamicJava*, `http://www.inria.fr/koala/djava/`

[24] Aubrey Jaffer, *SCM*, `http://www-swiss.ai.mit.edu/$\sim$jaffer/SCM.html`

[25] Aubrey Jaffer, *test.scm*, `http://ftp.swiss.ai.mit.edu/pub/scm/OLD/test.scm`

[26] Bruce A. Jognson, *test.scm*, `http://www.nmrview.com/swank/index.html`

[27] *JSR #000051: New I/O APIs for the Java Platform*, `http://java.sun.com/aboutJava/communityprocess/jsr/jsr_051_ioapis.html`

[28] Richard Kelsey, William Clinger and Jonathan Rees editors, *Revised^5 Report on the Algorithmic Language Scheme*, (2 November 1991). `http://www.swiss.ai.mit.edu/ftpdir/scheme-reports/r5rs-html/r5rs_toc.html`

[29] Richard Kelsey and Jonathan Rees, *A Tractable Scheme Implementation*, in *Journal of Lisp and Symbolic Computation*, 7:315-335 (1994). `http://www-swiss.ai.mit.edu/$\sim$jar/s48.html`

[30] Paul Kinnucan, *Java Development Environment for Emacs*, `http://sunsite.dk/jde/`

[31] Donovan Kolbly, *RScheme*, `http://www.rscheme.org`

[32] Robert Krawitz, Bil Lewis, Dan LaLiberte, Richard M. Stallman, Chris Welty, et al., *GNU Emacs Lisp Reference Manual*, Free Software Foundation, Cambridge, Massachusetts, edition 2.4b. `ftp://ftp.gnu.ai.mit.edu/pub/gnu/elisp-manual-19-2.4.2.tar.gz`

[33] Tim Lindholm and Frank Yellin, *The Java Virtual Machine Specification*, Addison-Wesley (September 1996). `http://java.sun.com/doc/books/vmspec/html/VMSpecTOC.doc.html`

[34] John K. Ousterhout, *Tcl: An embeddable command language*, in *The Proceedings of the 1990 Winter USENIX Conference*, pp. 133-146. `ftp://ftp.smli.com/pub/tcl/docs/tclUsenix90.ps`

[35] John K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley (1 May 1994).

[36] Project GNU, *Guile*, `http://www.gnu.org/software/guile/guile.html`

[37] Wolfgang W. Kuchlin and Jeffrey A. Ward, *Experiments with Virtual Threads*, Technical report, CIS Department, Ohio State University, Columbus, OH, September 1992. [Kuchlin and Ward, 1992]

[38] Michael G. Lehman, *HotTEA*, `http://www.cereus7.com`

[39] McCarthy, John. *History of LISP.*, ACM Sigplan Notices 13,8 (August 1978), `http://www-formal.stanford.edu/jmc/history/lisp.html`

[40] Microsoft Corporation, *JScript*, `http://www.microsoft.com/jscript`

[41] Microsoft Corporation, *Visual Basic*, `http://www.microsoft.com/vbasic`

[42] Microsoft Corporation, *ActiveX Scripting* (24 October 1996). `http://www.microsoft.com/intdev/sdk/docs/olescrpt/axscript.htm`

[43] Scott G. Miller, *LISC (LIghtweight Scheme on Caffeine)*, `ftp://ftp.gamora.org/pub/gamora/lisc`

[44] MIT Project on Mathematics and Computation, *MIT Scheme*, `http://www.swiss.ai.mit.edu/projects/scheme`

[45] John Neffenger, *VolanoMark*, `http://www.volano.com/benchmarks.html`

[46] Peter Norvig, *SILK - Scheme in Fifty KB (in Java)*, `http://www.norvig.com/SILK.html`

[47] Jonathan Rees, *Pseudoscheme*, `http://www.swiss.ai.mit.edu/ftpdir/pseudo`

[48] Jonathan A. Rees and Norman I. Adams IV, *T: A dialect of Lisp or, lambda: The ultimate software tool.* In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pp. 114-122 (1982).

[49] *JPython*, `http://www.jpython.org`

[50] *Rhino: JavaScript for Java*, `http://www.mozilla.org/rhino`

[51] Scriptics, *Jacl*, `http://dev.scriptics.com/software/java`

[52] Olin Shivers, *Cig—a C Interface Generator for Scheme 48*, http://www.swiss.ai.mit.edu/ftpdir/scsh/scsh-paper.ps

[53] Olin Shivers, *A Scheme shell*, To appear in the *Journal of Lisp and Symbolic Computation*. http://www.swiss.ai.mit.edu/ftpdir/scsh/scsh-paper.ps

[54] Olin Shivers and Brian D. Carlstrom, *The scsh manual*, MIT Laboratory for Computer Science (November 1995). http://www.swiss.ai.mit.edu/ftpdir/scsh/scsh-manual.ps

[55] Olin Shivers, *Supporting dynamic languages on the Java virtual machine*, (25 April 1996). http://www.ai.mit.edu/$\sim$shivers/javaScheme.html

[56] Jeffrey Mark Siskind, *Stalin - a STAtic Language ImplementatioN*, ftp://ftp.nj.nec.com/pub/qobi/stalin-0.8.tar.Z

[57] Guy L. Steele, *Common Lisp the Language, Second Edition*, Digital Press (May 1990) http://www.cs.cmu.edu/Web/Groups/AI/html/cltl/cltl2.html

[58] T. Suganuma, et al., *Overview of the IBM Java Just-in-Time Compiler*, http://www.research.ibm.com/journal/sj/391/suganuma.html

[59] Sun Microsystems, Inc., *Multithreaded Programming Guide*, http://www1.fatbrain.com/bookinfo/bookinfo.cl?theisbn=DM10002726 http://docs.sun.com/ab2/coll.45.13/MTP/@Ab2TocView?Ab2Lang=C&Ab2Enc=iso-8859-1

[60] Robert Tolksdorf, *Programming Languages for the Java Virtual Machine*, http://grunge.cs.tu-berlin.de/$\sim$tolk/vmlanguages.html

[61] Christian Queinnec, *PS^3I*, http://youpou.lip6.fr/queinnec/VideoC/ps3i.html

[62] Pinku Surana and Mark DePristo, *The Hotdog Compiler*, http://www.cs.northwestern.edu/$\sim$surana

[63] Michael Travers, *Skij*, http://alphaworks.ibm.com/tech/Skij

[64] John Vert, *Writing Scalable Applications for Windows NT*, Windows NT Base Group http://msdn.microsoft.com/library/techart/msdn_scalabil.htm

[65] Chris Walton, *LispkitLisp Compiler in Java*, (1997) http://www.dcs.ed.ac.uk/home/cdw/MyProjects/SECD/Applet/lispkit.html

[66] Arjuna Wijeyekoon, *MIT Scheme in Java*, http://web.mit.edu/arjuna/www/scheme/scheme.html