LISP – a Language for Internet Scripting and Programming

Timothy J. Hickey[*]
Peter Norvig[†]
Kenneth R. Anderson[‡]

Abstract

In this paper we argue that LISP can provide a powerful tool for web programmers, at both the novice and expert level, and that LISP has the potential to become a language of choice for writing both client-side and server-side internet programs. The syntactic and semantic simplicity of LISP enables non-experts to quickly master a basic level of LISP programming. Its higher order functions enable the implementation of a simple and elegant LISP interface to Java methods, fields, and constructors. With this interface, LISP provides full and simple access to the versatile and omnipresent Java class libraries. The conciseness of some dialects of LISP (e.g., Scheme) makes it relatively easy to implement compact LISP interpreters in Java and LISP-to-Java compilers. LISP can then be called from Java programs and Java-enabled browsers. The declarative nature of LISP allows one to design and implement simple, declarative interfaces to the Java class libraries, which allow one to create applets and client-server software which is much more concise and comprehensible than the same programs written in imperative languages such as Java or Javascript (for applet programming) or Java, Perl, C++, or C for client-server programs. Two additional fertile applications for LISP on the internet are debugging and scripting of Java code. In this paper, we provide small examples of all of these applications, describe our LISP implementation (SILK - Scheme In 50 KB), and ponder the future of LISP as a language for internet scripting and programming.

1 Introduction

LISP is poised to leap into the mainstream in a new role as a web programming language. The exponential growth in the WWW is due in large measure to the simplicity, in the eyes of the user, of the technology. To navigate the web one need only know how to read and

[*]Computer Science Department, Brandeis University, Waltham, MA
[†]Junglee Corporation, Sunnyvale, CA (currently at NASA Ames Research Center, Moffett Field, CA)
[‡]Bolt, Beranek, and Newman, Cambridge, MA

click. Creating a web page is only slightly more demanding due to the reliance on the syntactically simple HTML model and the use of a relatively small set of text formatting commands.

Applet programming has not made the same inroads into the public consciousness. One reason for the relative dearth of applets is the relative difficulty in learning Java (or Java Script). What LISP has to offer is a simple syntax and declarative semantics which can easily be mastered at an elementary level by novices, combined with a powerful abstraction mechanism that will appeal to programming professionals of all types, not just experienced LISP programmers.

In this paper we describe our initial efforts to extend LISP into a web programming language. We have selected Scheme as the LISP dialect, primarily due to its simplicity and its small size (which translates into short download time for a Scheme interpreter), and we have developed a web-based dialect of Scheme, called SILK [8]; and a declarative interface to the Java libraries, called JLIB [3].

1.1 Design Goals

To gain wide acceptance, the entire process of creating applets in LISP, debugging them, and adding them to a web page must be as simple as possible and must, in addition, result in applets that are at least as fast as Java applets.

1. **Easy access.** By implementing an integrated development environment for LISP as an applet, web denizens are able to start writing and running LISP programs and applets by simply visiting a web page. There is no software to install beyond the browser and no special skills to master before learning LISP. We have implemented a simple LISP interpreter applet (written in LISP) which allows users to peruse several example applets and to write and modify their own.

2. **Simple creation of applets.** By developing a high level declarative interface to the most commonly used parts of the Java libraries, we are able to provide a rapid entry path into applet writing for novice (and expert) programmers alike. The development of helpful online libraries of sample code and tutorials will also serve to further this goal. We have implemented a library (JLIB) which provides a declarative "Graphical User Interface" (GUI) toolkit. This toolkit is discussed in more detail below. We have used an early version of this library to provide an introduction to GUI design in an undergraduate course in Computer Graphics (CS155, Brandeis University, Spring 1998). After one 50 minute lecture the students were able to implement fairly sophisticated graphical applications in Scheme. To reach the same level of proficiency in Java in the same course required 10 lectures. We are currently using this library in an "Introduction to Computers" course (CS2a, Brandeis University, Autumn 1997, Autumn 1998) for non-computer science majors. We plan to spend four weeks teaching the fundamentals of programming and GUI design using Scheme as a first language. The advantage of teaching Scheme over Java at the introductory level is that the syntax and semantics of Scheme can be covered fairly completely in a few lectures,

leaving ample time to explore the GUI libraries and other interesting topics. Morover, the simple interface to Java allows us to introduce the graphics constructors and methods at the very beginning along with the arithmetic constants and operators. In contrast, it usually requires an entire semester to provide an introduction to Java, and most of the "Java as a first language" texts don't begin to cover the java.awt library until late in the book, if at all.

3. **Simple debugging.** This is an area where AI techniques could be quite helpful. Detecting and explaining common syntax and runtime errors is a crucial step in teaching a new language. Tools which provide this type of support could increase the interest in this language in the mainstream. Our current debugging support is minimal and provides only the basic commands: step, skip, and continue.

4. **Simple incorporation in web pages.** There are two approaches here. One is to develop a LISP-to-bytecode compiler which compiles LISP applets directly to class files (or indirectly through Java). This requires LISP applet writers to download the compiler (or change the security restrictions on a compiler applet). The other approach is to write a LISP interpreter applet in which the LISP program to be evaluated is passed as a parameter. We have concentrated on the latter, and have developed a threaded interpreter-based applet. We are presently working on compiler-based applets.

5. **Fast download and execution.** By using a LISP-to-bytecode compiler, we can in principle create applets which are as fast or faster than those written directly in Java, but this requires the applet writer to have access to a Java Development Kit since such a compiler must read and write local files and so will not run as an applet. We have opted instead to develop a LISP interpreter applet written in Java. Since the LISP interpreter is relatively small, we can attain reasonable download times, and by using compiler technology in the interpreter, we attain respectable execution times for interpreted applets. For example, the LISP interpreter applet itself requires about 8 seconds to download and initialize on a 200 Mhz Mac PPC running Netscape 4.04 under Linux with a T1 internet connection). It is possible to combine these two approaches by providing access to compiled libraries of Scheme procedures, which can be stored in the class archive and used by the applets. We have implemented this approach for the JLIB library mentioned above.

1.2 The Primitive LISP-Java Interface

The LISP-Java interface we have implemented is based on two procedures:

```
(constructor            CLASSNAME ARG1TYPE .... ARGNTYPE)
(method      METHODNAME CLASSNAME ARG1TYPE .... ARGNTYPE)
```

The `constructor` procedure is given a specification of a Java constructor (classname and argument types) and returns a procedure implementing that constructor. The `method` procedure accepts a specification of the method (methodname, classname, and argument types) and returns a procedure implementing that method.

For example, to determine whether a large number is probably prime with an error of about $1/2^n$, we can use the "isProbablePrime" method of the "BigInteger" class (Note that the confidence limit, "n", is the last parameter of this method):

```
(define BigInteger
    (constructor "java.math.BigInteger" "java.lang.String"))
==> BigInteger

(define isProbablePrime
    (method "isProbablePrime" "java.math.BigInteger" "int"))
==> isProbablePrime

(isProbablePrime (BigInteger "1231231231231231231231") 10)
==> false
```

In this short session we have used the java.math package to demonstrate that the 22 digit number above is probably composite. (Note that we don't know any of its factors.)

We also need to be able to access and modify fields of objects. This is done using the following two procedures:

```
(field-getter FIELDNAME CLASSNAME)
(field-setter FIELDNAME CLASSNAME)
```

The first returns a procedure for accessing the field's value, and the second returns a procedure for modifying the field's value.

So, for example, if "Pair" is a class in a package "silk" with a field "first",

```
public class Pair {
  public Object first, rest;
  public Pair(Object car, Object cdr) {
    first = car; rest = cdr;
  }
}
```

then we could define and use a constructor of Pairs and a getter and setter of "first" as follows:

```
(define mycons
  (constructor "silk.Pair" "java.lang.Object" "java.lang.Object"))
(define mycar     (field-getter "silk.Pair" "first"))
(define mysetcar  (field-setter "silk.Pair" "first"))

(define a (mycons 1 2.5))
```

```
(display (mycar a))
(mysetcar a "hi")
(display (mycar a))
```

Its interesting to observe that the field-getter and field-setter procedures can be defined using the "method" and "constructor" procedures and the "java.lang.reflect" package, as follows:

```
(define getClass (method "getClass" "java.lang.Object"))
(define getField
  (method "getField" "java.lang.Class" "java.lang.String"))
(define getFieldValue
  (method "get" "java.lang.reflect.Field" "java.lang.Object"))
(define setFieldValue!
  (method "set" "java.lang.reflect.Field" "java.lang.Object"
        "java.lang.Object"))

(define classForName
  (method "forName" "java.lang.Class" "java.lang.String"))
(define field-getter
  (lambda (class-name field-name)
    (let ((field (getField (classForName class-name) field-name)))
      (lambda (object) (getFieldValue field object)))))
(define field-setter
  (lambda (class-name field-name)
    (let ((field (getField (classForName class-name) field-name)))
      (lambda (object value)
      (setFieldValue! field object value)))))
```

These procedures provide access to both instance and static variables. In the former case they use the first parameter, object, to specify the instance variable, in the latter case, the object parameter is ignored.

2 Declarative GUI programming

One of the most attractive features of LISP as a tool for building Graphical User Interfaces is that it supports a declarative style of GUI building in which the expression which creates a window has the same structure as the window itself. To attain this declarative simplicity we have implemented a high level interface to the Java Abstract Windowing Toolkit (AWT).

This interface allows for declarative creation of all standard components: window, label, textarea, textfield, button, choice, etc. For example, to create a button with the label "Go", or to create a choice of several numbers, one evaluates the following expressions.

```
(define b (button "Go"))
(define c (choice 1 2 3 4 5 10 25 50 100))
```

To provide a declarative layout mechanism we have implemented four layout methods:

1. `(window name height width c1 ... cn)` – create a window with the given name, height and width which contains the specified components c1,c2, ..., cn.
2. `(row c1 c2 ... cn)` – create a panel in which the components c1, ..., cn are arranged horizontally.
3. `(col c1 c2 ... cn)` – create a panel in which the components c1, ..., cn are arranged vertically.
4. `(grid rows cols c1 ... cn)` – create a panel in which the components c1, ..., cn are arranged in a 2D grid with the specified number of rows and columns. The components are placed in the cell from left to right, from the first row through the last. The grid cells all have the same shape and are just large enough to accomodate any of the components.

Event handling is done using a procedure `(pad comp proc)` which takes a component `comp` and a procedure `proc` of one argument and returns a component which responds to an action event e by calling the procedure `proc` on e. Thus, to create a "hello world" window with hide button, we simply evaluate the following expression. Observe how this expression has the same form as the window it creates and that the action appears with the component it is attached to.

```
(define w
  (window "hello" 200 200
     (col
        (label "Hello World")
        (pad (button "hide") (lambda (e) (hide w))))))
(show w)
```

Our final GUI abstraction is to introduce procedures for reading and writing strings and Scheme terms on components. This is done with a group of procedures. The two most commonly used are `readExpr` which reads the string labelling a component and parses it into a LISP term, and `writeExpr` which writes a LISP term on a component.

```
(readExpr component)
(writeExpr component expr)
```

We can put these together to create a GUI for a program to compute your Body Mass Index, as shown in Figure 1. (Note: this index is your weight in kilograms divided by your height in meters squared. It should be between 20 and 25.)

A `bmi-panel` consists of three components arranged vertically. First is a 2x2 grid containing the height and weight labels and textfields, next is the button for computing the BMI, and finally is the textfield where the BMI will be displayed. Observe that the button is associated with an action using the "pad" procedure. Figure 1 shows the result of evaluating this program using the SILK interpreter using Netscape 4.06 on an SGI Indy. Note that the SILK interpreter GUI was written in SILK itself.

This high level interface is admittedly limited, but it is ideal for the novice user or for an experienced user wanting to quickly implement a prototype. When more control over the layout is desired, one can either resort to the primitive LISP-Java interface to directly import procedures from the Java AWT (or any other Java windowing toolkit, such as Swing), or one can develop a more sophisticated high level interface.

3 LISP Applets

There are two methods for embedding LISP applets into web pages. The first method, which we currently use, is to implement a Scheme interpreter in Java, and to then create a Java applet (say of class"lisp.LispApplet") which accepts a program and an expression as applet parameters, creates the Scheme interpreter, loads the program, and evaluates the expression. For example, the following is a sample applet tag for running the Body Mass Index program from the previous section as an applet.

```
<applet height = 800 width = 600
        code = "lisp.LispApplet.class">
<param name="program"    value="BMI.scm">
<param name="expression" value="(add this-applet (bmi-panel))">
</applet>
```

The expression `(add APPLET COMPONENT)` adds the component to the applet, the variable `this-applet` is initialized by the LISP interpreter applet. For this approach to be practical, the LISP interpreter applet must be fairly small to enable short download times, i.e., lisp.LispApplet.class must be relatively small. Our current interpreter applet is around 50Kb of bytecode and download times vary from 5-45 seconds depending on the Java Virtual machine, the hardware, the network connection, and the operating system.

The second method for embedding LISP applets into a web page is to use a LISP-to-Java compiler. In this case, the LISP applet can be compiled to Java, and then further compiled to a Java byte code class file and so can be installed on a web page just as any other applet is:

```
<applet height = 800 width = 600
        code = "lispuser.BMI.class">
</applet>
```

Although this method requires more work, it has the potential to provide more efficient applets and greatly decreased download times. Another advantage of this approach is that by compiling to Java, we are able to make use of the latest Java compilation technology.

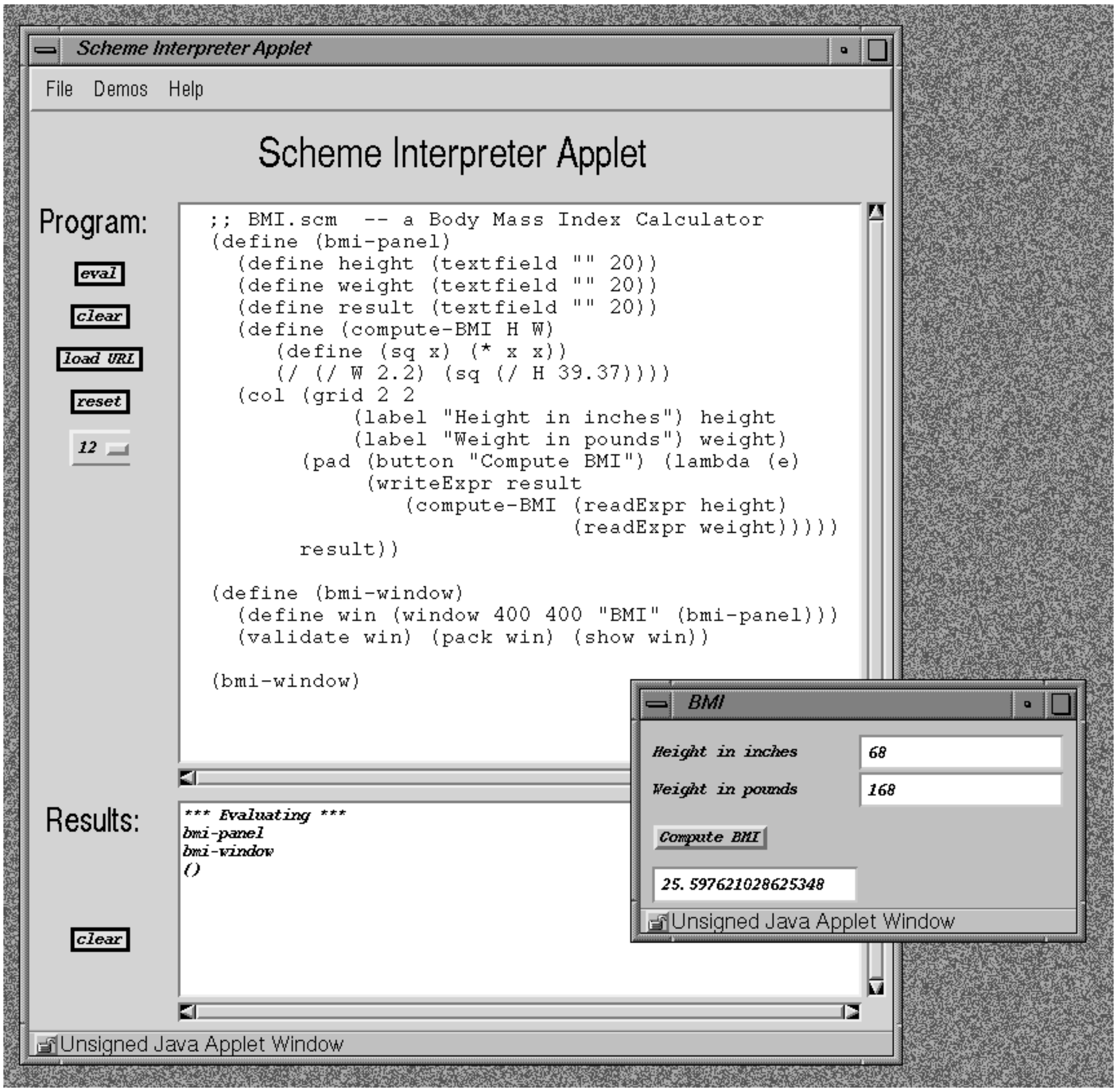

Figure 1: The BMI GUI interpreted with the SILK interpreter

In both approaches, LISP provides a simple alternative to Java and Java script as an applet-writing language. Ideally, we would like to have a new HTML (or XML) tag for invoking lisp applets, e.g.,

```
<LISPapplet height = 800 width = 600
    prog = "../demos/Grades.scm"
    expr = "(run-code this-applet)">
```

or a Java Script style tag which would allow the program to be included directly in the web page. These extensions could be achieved by building a LISP interpreter into the current browsers.

4 Interactive Java Debugging and Scripting

A fertile area for LISP in the internet mainstream is debugging and testing. It is often convenient in Java to put testing code in a static method, such as `main()`. Typically such test code is not interactive. An interactive test loop can require a significant amount of code that can be unique for each class. While a testing tool, such as JUnit [1] may make writing test cases easier, it may not be interactive enough to fully diagnose a problem.

Below we give an example of a session in which the user has tested some of the methods and constructors from a Java program. The Java code in this example was written by one of the authors as part of an Interval Arithmetic Constraint Solver.

The main point we wish to illustrate here is that LISP can be viewed as a powerful interactive debugging tool for general Java programs. In this example below, we first import a constructor and three methods into LISP.

```
;;;;;; MAP JAVA METHODS AND CONSTRUCTORS INTO LISP
;; create a table which stores variable-interval pairs
(define RealIntervalTable
   (constructor "ia_parser.RealIntervalTable"))

 ;; parse a string into an internal representation of a constraint
(define parseString
   (method "parseString" "ia_parser.Parser" "java.lang.String"))

;; store the variables of a constraint in an interval table
(define storeVariables
   (method "bindVars" "ia_parser.Exp"  "ia_parser.RealIntervalTable"))

;; use the constraint to narrow the intervals it contains
(define narrow (method "narrow" "ia_parser.Exp"))
```

After defining these procedures, we can interactively call these imported procedures and examine the results.

```
;;    EXPRESSION                      RESULT
(define c
  (parseString "x = cos(x);")) ==> c
c                              ==> "x = cos(x);"
(define T (RealIntervalTable)) ==> T
T                              ==> ()
(storeVariables c T)           ==> ()
T                              ==> (<x -> [-inf,inf]>)
(narrow c)                     ==> true
T                              ==> (<x -> [-1,1]>)
(narrow c)                     ==> true
T                              ==> (<x -> [0.540302,1]>)
(narrow c)                     ==> true
T                              ==> (<x -> [0.540302,0.857553]>)
```

In this example, parseString creates an interval expression, RealIntervalTable creates an object for storing variable-interval bindings, storeVariables initializes the table to contain the variable `x` in the constraint `x=cos(x);` with its current (most general) binding `[-inf,inf]`, stating that x can be any real number. The narrow procedure then attempts to shrink the interval for x with out removing any solutions to the constraint `"x=cos(x)`. If the interval for `x` becomes empty (meaning the constraint has no solutions), then narrow returns `false`, otherwise it returns `true`. Observe that the first call to narrow reduces the interval for `x` to `[-1,1]`, which is the range of cos. The next narrowing raises the lower bound, and the next lowers the upper bound, this process can be repeated about 50 times until a fixed point is reached, and the resulting interval is guaranteed to contain any solution to `cos(x)=x` (assuming that the narrowing procedure has been written correctly). This example shows how a simple Scheme interpreter combined with the Scheme-Java interface, provides a powerful tool for interactively testing general Java programs.

Another attractive use of LISP is in writing scripts to implement applications by glueing together previously developed Java programs with a little bit of LISP. For example, in Figure 2 we show how a GUI for an interval arithmetic solver can be constructed in a few lines of Scheme using the four imported procedures from the previous example. In this example, we use the declarative GUI building library discussed above to build a simple graphical interface which allows the user to type a constraint into a textarea and then push a button to (iteratively) narrow the constraint, the resulting interval table is written into another textarea.

```
;;;;;; Script a narrowing procedure
(define tab (RealIntervalTable))
(define (narrow-expr str)
  (define con (parseString str))
  (storeVariables con tab)
  (if (narrow con) tab "NO SOLUTION"))
;;;;;; Create GUI components for user I/O
(define solvewin (window 300 300 "IAsolver"))
(define constraint (textarea 10 60))
(define variables (textarea 5 60))
;;;;;; Lay out GUI components and attach actions
(add solvewin
  (col constraint
       (pad (button "solve") (lambda (e)
            (writeExpr variables
               (narrow-expr (read_from constraint)))))
       variables))
(validate solvewin) (pack solvewin) (show solvewin)
```

Figure 2: An Example of Scripting Java using LISP

5 SILK design issues

The Scheme implementation we use is SILK, for "Scheme In about 50 K". The original versions up to SILK 1.0 were developed by Peter Norvig. After Peter made this implementation available on the web several people made useful (and sometimes almost identical) extensions. SILK 2.0 compiles Scheme syntactic expressions into objects of class `Code` that can be more efficiently evaluated. Version 2.0 was started by Peter Norvig and completed by Tim Hickey. The remainder of this section describes the design issues involved in the two implementations.

The initial version of SILK was written in about 20 hours with about 650 lines of code. The primary goals were to develop a LISP that was small, fast to load (even over the web), easy to understand and modify, and that could interface to Java. SILK expanded to about 50KB of Java code over the next few months as it was extended to pass all of the tests in Aubrey Jaffer's online r4rstest.scm test suite [4] which tests Scheme compliance with the R4RS standard.

The class structure for SILK 1.0 is shown in Figure 3.

The SchemeUtils class contains the basic LISP procedures (e.g., car, cdr, cons) as static methods. It is inherited by the five top level classes as a convenience (so that one can write `car(x)` rather than `SchemeUtils.car(x)`). The class Scheme is the interpreter class and it has been designed so that one could easily swap several interpreters in; perhaps a simple

- SchemeUtils
  - Scheme
  - Environment
  - Pair
  - InputPort
  - Procedure
    - Primitive
    - JavaConstructor
    - JavaMethod
    - Continuation
    - Closure
      - Macro

Figure 3: Class Structure for SILK 1.0

interpreter, a tail-recursive interpreter, and an interpreter that supports full `call/cc`, as in PAIP [7]. The Scheme primitives are stored in their own class so that it would be easy to switch from R4RS to R5RS [5] Scheme primitives.

Once these class design issues are out of the way, the rest is pretty easy. In order to build a Scheme interpreter, you basically need six things:

1. **Read and write.** The reader goes in the `InputPort` class. For output, most of the functionality resides in a method, `SchemeUtils.stringify`, to convert an object to a printable string. Once we have the string, printing is trivial, so one can use the existing Java `PrintWriter` class, a new class for `OutputPort` isn't needed.

2. **Eval and apply.** The `eval` method goes in the `Scheme` class. There's also an `apply` method there, but all it does is make sure you're applying a procedure and then dispatches to the `apply` method in each subclass of `Procedure`. Unfortunately, we have to repeat the code for applying a `Closure` (a user-defined Scheme procedure) within the code for `eval`. This is necessary only to make sure that `eval` is properly tail recursive. If Java were tail recursive, we wouldn't need to do this.

3. **Memory management**. Java handles this automatically.

4. **Run-time stack**. SILK uses the Java run-time stack. This means it don't have full continuations, just throw-like ones, and it doesn't have good debugging capabilities. But it does make things easier than implementing a separate Scheme stack. However, SILK is properly tail recursive: calls in the tail position do not grow the stack.

5. **Primitive functions** The class `Primitive` provides for the application of primitive procedures like `list`, `pair?`, and `write`. One possible implementation strategy is to have each primitive be an anonymous inner class:

```
new Primitive("list", env, 0, n) {
   public Object apply(Object args) { return args; }};

new Primitive("null?", env, 1) {
   public Object apply(Object args)
     { return truth(first(args) == null); }};
```

The other possibility is to have a big switch statement. Since loading 150 inner classes might be slow, SILK uses the big switch. This means a lot of busy work, and breaking up the definition of each primitive into three places: first there is a long list of final static ints for the switch labels, next all procedures need to be initialized (since procedures are first class objects in Scheme), and the final part is the big switch that defines the semantics of each primitive.

```
final static int ... LIST = 16,..,NULLQ = 23, // define constants
...
public static Environment installPrimitives(Environment env) {
 int n = Integer.MAX_VALUE;
    env.defPrim("list", LIST, 0,n)// list has 0 to n arguments
       ...
       .defPrim("null?",NULLQ, 1) // null? has 1 argument
       ...;
}
public Object apply(Scheme interpreter, Object args) {   ...
   Object x = first(args);
   switch(idNumber) {       ...
   case LIST: return args;
   ...
   case NULLQ: return truth(x == null);
   }
}
```

6. **Primitive data types.** Scheme has a dozen or so data types (depending on how fine distinctions you make) that are visible to the programmer. SILK implements these data types using the simplest possible existing Java types wherever possible (see Figures 4 and 5). So, for example, a Scheme vector is implemented as a `Object[]`, not a `Vector`, because Scheme vectors don't need to add and delete elements. Using existing Java types means that we can't call `obj.display()`, because there is no `display()` method on these existing Scheme classes, and of course you can't add a method to an existing class in Java. So instead, we will have lots of `static` methods in the `SchemeUtils` class. For numbers, version 1.0 had only one type: Double. The Integer type was added in version 2.0 so that the exact/inexact distinction could be correctly implemented.

Scheme Type	Java Type (version 1)	Java Type (version 2)	Notes
pair	Pair	Pair	Java has no equivalent to use
empty list	null	null	We could have `Pair` and `Empty` as subclasses of `List`, or `EMPTY` could be a static var. But it's easiest to have `null` for `()`. This means only static methods on lists. (It also means we can't use Hashtables to store Scheme objects!)
boolean	Boolean	Boolean	Simple enough.
inexact number	Double	Double	One could use `BigDecimal`, but R5RS doesn't requuire it.
exact number	Double	Integer	One could use `BigInteger`, but R5RS doesn't requuire it.
string	char[]	char[]	Can't be `String` (they're immutable) unless we want to give up on implementing `string-set!`. Could be `StringBuffer`, but `char[]` is simpler and sufficient.
character	Character	Character	Simple enough
symbol	String	**Symbol**	Strings (once interned) have the right properties for symbols, but it gets confusing when we interface to Java code that uses Strings for other things. So version 2 introduces a Symbol class, which makes things less confusing, and allows global variable lookup to be faster.
vector	Object[]	Object[]	Because `Vector` is more powerful than is needed.
Procedure	Procedure Primitive Closure SchemePrimitives Macro Continuation JavaMethod JavaConstructor	Procedure Primitive Closure SchemePrimitives Macro Continuation JavaMethod JavaConstructor	the abstract superclass. a Scheme primitive. a user-defined function. Scheme code loaded at start-up. a code expansion "Closure". this is what you get from `call/cc`. encapsulates a method. encapsulates a constructor

Figure 4: Java Implementation of Scheme Types

Scheme Type	Java Type (version 1)	Java Type (version 2)	Notes
input port	InputPort	InputPort	It would have been simpler to use `Reader`, the code for `read` needs to be put somewhere, and hence this class.
output port	PrintWriter	PrintWriter	We could have had a class to hold the method `stringify`, but its not really necessary to have another class.
interpreter	Scheme	Scheme	The class `Scheme` implements `eval`, `apply`, and a few other methods. Note you can instantiate several different Schemes at once.
environment	Environment	GlobalEnvironment LambdaEnvironment	Implemented using Object[] Implemented as list of lists.
continuation	Continuation	Continuation	this builds a `Continuation` which, when applied, throws a `Runtime Exception`. It then sets up a `try catch` block that catches that identical exception only.
error	RuntimeException	RuntimeException	RuntimeExceptions are used because they do not have to be declared and so the SILK code will not be cluttered up with `throws` clauses.
symbol table	(none)	HashTable	In version 1, Scheme symbols are represented as interned Strings. In version 2, we maintain our own String $\rightarrow$ Symbol Hashtable as a static variable in Symbol.
variable	(none)	(none)	Variables are referred to implicitly by their position in Environments.

Figure 5: Java Implementation of Scheme Types (continued)

6 Comparison of Scheme Implementations

Scheme is a powerful but tiny language. The entire language manual, including it's denotational semantics, is 50 pages long. A Scheme interpreter is easy to implement. Also, techniques for compiling Scheme programs are well known. These facts have lead to many Scheme implementations with a broad range of sizes and performance characteristics. For example, Kelsey and Rees found seven Scheme implementations all less than 14,000 lines, and 3 implementations that ranged from 25,000 to 120,000 lines [6]

Each language that Scheme is implemented in provides both opportunities and challanges. For example, when implementing Scheme in C, one must work hard to provide garbage collection and tail recursion. (call-with-current-continuation) is often the hardest thing to implement in any base language.

One advantage of Java is that garbage collection is provided. One disadvantage is that it is not possible to take advantage of the compact representations for some Scheme data types. For example in Scheme (or LISP in general), small objects such as characters and small integers are often represented as an "immediate" pointer sized object. A list cell, a cons, can be represented as two words.

In Java, this is not possible. Any Scheme object must be represented as a Java object. Currently, in JDK 1.1.6, a Java object requires at least 12 bytes of storage. For example, a pair, which in SILK is represented by an instance of the class Pair, requires 24 bytes.

The following subsections describe the Scheme-in-Java dialects we are aware of, and contrasts their capabilities. Figure 6 shows the relative sizes of the implementations. We use raw line counts because they often contain both code and documentation. Silk is written in a rather compressed style, so the table may be unfair to implementations written in another style or with other goals in mind.

6.1 Silk 1.0

Silk 1.0 was designed to be a tiny Scheme environment. It is programmed in a style that makes extensive use of static methods so the code looks relatively Scheme-like. Existing Java classes were used wherever possible. Java null is used as the empty list. Extensions to interface to Java are minimal: (constructor) and (method).

Implementation	Java files	lines	Scheme files	lines
Silk 1.0	12	1905	0	0
Silk 2.0	20	2778	0	0
Skij	28	2523	44	2844
Jaja	66	5760	2	173
Kawa	273	16629	14	708

Figure 6: Scheme implementation statistics

6.2 Silk 2.0

Silk 2.0 provides a higher performance evaluator by splitting eval() into two steps: first compile the meaning of the object, and then call the eval method on the meaning. This gains efficiency because the compilation step is done only once on code that can be evaluated many times. In practice, Silk 2.0 is about twice as fast as 1.0.

```
      /** Evaluate an object x in an environment env.  **/
      public static Object eval(Object x, Environment env) {
        return Code.eval(Code.toCode(x,env), Frame.EMPTY);
 }
```

The main design issue is how to provide tail recursion. This is done by letting `Code.eval()` return either an `Object` or a `Code`, and iterating if a `Code` is returned.

```
static Object eval(Object expr, Frame f) {
  if (expr instanceof Code) {
    expr = ((Code)expr).eval(f);
    while (expr instanceof Code)
      expr = ((Code)expr).eval(Frame.EMPTY);
  }
  return expr;
}
```

Thus the inner loop of the interpreter is replaced by a method dispatch. Currently, a relatively small number of `Code` inner classes are defined. Further performance can be gained by generating more refined classes to handle common special cases more efficiently.

6.3 Skij

Skij is a Scheme advertised as a scripting extension for Java. It is similar in capabilities to Silk 1.4, but has more extensive Java support including (peek) and (poke) for reading and writing slots, (invoke) and (invoke-static) for invoking methods, and (new) for constructing new instances of a Java class. The operations (new), (invoke), and (invoke-static) are generic. The appropriate Java method is looked up at runtime based on all of its arguments. While Silk users have experimented with similar generic operations, they are currently not part of the standard distribution.

Many procedures are provided in Scheme code, some of which simply invoke an underlying Java primitive.

6.4 Jaja

Jaja is a Scheme based on the Christian Queinnec's wonderful book "Lisp in Small Pieces"[11]. It includes a Scheme to Java compiler written in Scheme, because it requires only 1/3 the code of a Java version. Compared to Silk, Jaja is written in a more object oriented style.

Like Silk, Jaja uses a super class (Jaja in Jaja, and SchemeUtils in Silk) to provide globals and utility functions. Unlike Silk, in Jaja, each Scheme type has one or more Java classes defined for it. Also, in Jaja, `nil` is represented as an instance of the class `EmptyList`, while in Silk it is represented by null. All Jaja objects are serializable.

6.5 Kawa

Kawa is an ambitious Scheme implementation. It includes a Scheme to Java byte code compiler. Each function becomes a Java class compiled and loaded at runtime. It is the largest of the Scheme-in-Java implementations.

7 Future

In this section we outline several extensions we hope will appear in future versions of SILK. They should enhance the power of LISP/Java integration.

7.1 Generic functions

As described above, the (constructor) and (method) procedures provide access to Java methods. However, these methods are not generic. This means that the two get() methods for Java's Field and Hashtable classes must be named apart:

```
(define getFieldValue
  (method "get" "java.lang.reflect.Field" "java.lang.Object"))
(define getHashtable
  (method "get" "java.util.Hashtable" "java.lang.Object"))
```

A more natural solution is to allow (get) to be a generic function, as in Common Lisp. The appropriate method to invoke would be chosen based on the argument types at runtime. The Skij Scheme implementation already has this capability. We have experimented with such generic functions based on Tiny CLOS. One important extension is that both Java methods and Scheme methods can be added to a generic function.

7.2 Compilers

Compiling Scheme to Java is essential for increasing performance of Scheme code. The Jaja and Kawa Scheme implementations already provide interesting compiling capabilities. We have experimented with a simple compiler that is essentially a partial evaluation of the Silk 1.4 interpreter. Given a file of Scheme definitions it produces source code for an equivalent Java class. The class can then be compiled and used in place of the Scheme code. This simple compiler increased performance by a factor of two and reduced applet loading time.

7.3 Meta Scripting

A simpler variant of a compiler is to write Scheme procedures that generate new Java classes from old ones. Since SILK has access to the reflective description of a Java class, it can be used as a template to generate a new class with additional capabilites. For example, supose we wanted to trace the get() and put() methods of a Hashtable. The form:

```
(define-class ("myPackage" "myHashtable")
  (extends "java.util.Hashtable")
  (methods
    (trace "get")
    (trace "put")))
```

could produce a new subclass with this capability:

```
import trace.Trace;
package myPackage;

public class myHashtable extends java.util.Hashtable {
  public Object get(Object key) {
    Trace.print(this + "get(" + key + "): ")
    Object result = super.get(key);
    System.out.println(result);
    return result;
  }
  public void put(Object key, Object value) {
    Trace.print(this + "put(" + key + ", " + value + ")");
    super.put(key, value);
  }
}
```

In SILK, the class Primitive that implements about 150 Scheme primitives can be automatically generated in this fashion. Each primitive is described by a one line Scheme form. The forms are then woven together to construct the Java code for the Primitive class. The necessary bookkeeping and glue code, described above, is generated automatically. This primitive information can also be used by a LISP-to-Java compilier.

7.4 Network citizenship and Java integration

With Java as the implementation language, there are many opportunites that SILK can take advantage of in the future. These include Thread support, Serialization, and Networking.

References

[1] Kent Beck and Erich Gamma, *Test-infected: Programmers love writing tests,* Java Report, Vol 3, No. 7, p.51 - 56, 1998.
`http://members.pingnet.ch/gamma/junit-10.zip`

[2] Per Bothner, *Kawa, the Java-based Scheme System*
`http://www.cygnus.com/ bothner/kawa.html`

[3] Tim Hickey (Project Leader) *JLIB: A Declarative GUI-building library for SILK*
`http://www.cs.brandeis.edu/ tim/Packages/jlib/jlib.html`

[4] Aubrey Jaffer *R4RS Scheme test*
`ftp://ftp-swiss.ai.mit.edu/pub/scm/r4rstest.scm`

[5] Richard Kelsey, William Clinger, and Jonathan Rees (Editors). *Revised(5) Report on the Algorithmic Language Scheme.* 1998.
`www.math.grin.edu/courses/Scheme/r5rs-html/r5rs_toc.html`

[6] Richard A. Kelsey, and Jonathan A. Rees, *A Tractable Scheme Implementation*, Lisp and Symbolic Computation, 7, 4, p. 315-336, 1994.

[7] Peter Norvig *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp* Morgan Kaufmann, 1992.

[8] Peter Norvig (Project Leader) *SILK: Scheme in Fifty K*
`http://www.norvig.com/SILK`

[9] Michael Travers *Skij*, IBM alphaWorks archive
`http://www.alphaworks.ibm.com/formula/Skij`

[10] Christian Queinnec *JaJa: Scheme in Java*
`http://www-spi.lip6.fr/ queinnec/WWW/Jaja.html`

[11] Christian Queinnec *Lisp in Small Pieces,* Cambridge University, Cambridge, 1996.