SILK - a playful blend of Scheme and Java

Kenneth R. Anderson, BBN Technologies, Cambridge, MA
KAnderson@bbn.com

Timothy J. Hickey, Brandeis University, Waltham, MA
tim@cs.brandeis.edu

Peter Norvig, NASA Ames Research Center, Moffett Field CA 94035-1000
peter@norvig.com

If we left out the prompt, we could write a complete Lisp interpreter using just four symbols:

```
(loop (print (eval (read))))
```

Consider what we would have to do to write a Lisp (or Java) interpreter in Java
[after PAIP p. 176].

Abstract

SILK (Scheme in about 50 K) is a compact Scheme implemented in Java. It is currently in its fourth implementation. The first version implemented a nearly R^4RS Scheme in Java, but its access to Java was awkward. The current version has altered SILK's syntax and semantics slightly to better integrate with Java. This has simplified its implementation, and made SILK an effective Java scripting language, while preserving its Scheme flavor. SILK applications, applets and servlets are suprisingly compact because they take advantage of Scheme's expressiveness. Because SILK is interactive and has reflective access to Java, it provides a view into a Java application that few Java programmers have seen. For example, it easily reveals a security issue involving inner classes. SILK is an Open Source project at http://silk.sourceforge.net

Introduction

Java, is a popular language to implement other languages in. For example, [RT] identifes about 130 languages implemented in Java, in the past five years. Currently, 15 of those are Lisp related languages, with eight of them focused on Scheme.

There are many reasons for this. Java is a simple object oriented language with syntax like C, single inheritance like Smalltalk, and garbage collection like Scheme. Java provides standard libraries that support web programming. It has the goal of being portable, with the motto "Write once, run anywhere". Java also has the biggest and noisiest bandwagon.

Not only is Java a good target implementation language, but because Java provides no syntactic extension mechanism, Java applications can benefit from extension minilanguages. Scheme-like extension languages, such as SILK, can be especially effective because they can be a fertile soil for such minilanguages.

Here we describe our experience with SILK, a Scheme implementation in Java. SILK is currently in its fourth implementation. SILK started as a small Scheme implementation with limited access to Java. Each method and constructor had to be individually pulled accross the Scheme/Java barrier.

Now, Java metaobjects (classes, fields, methods and constsructors) can be manipulated directly using a simple syntactic extension. SILK matches the Scheme type system to existing Java types, where it can. Also, SILK's syntax has been extended to allow for constants of all Java numeric types. This reduces the overhead of invoking Java methods from SILK, and reduces the ambiguity of dynamic method invocation.

The result is a Scheme dialect with direct, almost transparent, access to its implementation language, Java. In a C implementation of Scheme, such as Guile [Guile] or STK [STK], after providing a Scheme implementation, one also provides extensions such as a Foreign Function Interface, a library of tools, such as SLIB, and an object oriented extension, such as GOOPS or STKLOS. In SILK, Java provides these extentions directly, simply by getting your CLASSPATH right.

In the remainder of the paper we describe the main features of SILK programming, and the main features of SILK's implementation.

SILK programming

SILK programming is like programming in Scheme with direct access to any Java class. The following sections describe the main features of SILK programming.

Scheme/Java Interface

To access Java, first, the `(import)` procedure can be used to simplify the naming of classes, much as the `import` statement does in Java:

```
> (import "java.awt.Dimension")  ;; Import a single class.
#t
> (import "java.util.*")         ;; Import all the classes in a package.
#t
```

Once a class or package has been imported, a class can be named by its name relative to the `(import)`:

```
> (class "Dimension")
class java.awt.Dimension
> (class "Hashtable")
class java.util.Hashtable
```

Imports accumulate throughout the SILK session, and ambiguous class names cause an error.

Secondly, variables containing "." or "$" are used to name Java methods, constructors, fields, and classes. This syntax provides full dynamic access to all Java classes. Here are the variable construction rules:

Type	Rule	Example

Class	className ".class"	`> Dimension.class`
		`class java.awt.Dimension`

Constructor	className "."	`> (define d (Dimension. 100 200))`
		`java.awt.Dimension[width=100,height=200]`

Instance Method	"." methodName	`> (.getWidth d)`
		`100.0`

Instance Field	"." fieldName "$"	`> (.width$ d)`
		`100`

Static Method	className "." methodName	`> (System.getProperty "java.version")`
		`"1.2.2"`

Static Field	className "." fieldName "$"	`> (.println System.out$ "Hello Sailor!")`
		`Hello Sailor!`
		`#null`

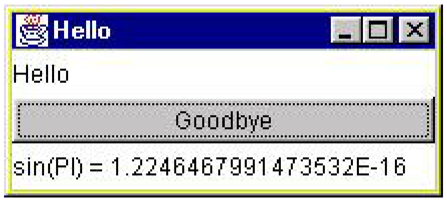
For an instance field, an accessor function is returned that takes an instance as its argument. Static fields are treated as global variables so they can be assigned to using `(set!)`:

```
(set! Frog.count$ (+ Frog.count$ 1))
```

If SILK can't find a Java object with the appropriate name, the variable is treated as a normal Scheme variable.

Comparison with Java

The following code creates a window with a close button, that looks like this:



In the exhibit below, the SILK version is on the left, and its equivalent Java version is on the right. Java constructors, instance and static methods, and instance and static fields are shown in **bold**.

```
;; Silk example
(import "java.awt.*")
(import "java.awt.event.*")


(define win (Frame. "Hello"))
(define b (Button. "Goodbye"))
(.add win (Label. "Hello")
      BorderLayout.NORTH$)
(define p (Label.
  (string-append "sin(PI) = " (.toString
    (Math.sin  Math.PI$)))))
(.add win b)
(.add win p  BorderLayout.SOUTH$)
(.addActionListener b
  (Listener. (lambda (e)
    (.hide win))))


(.pack win)
(.show win)
(.println System.out$ "Done")
```

```
/* Java example */
import java.awt.*;
import java.awt.event.*;
public class Demo {
static public void main(String[] args) {
  final Frame win = new Frame("Hello");
  Button b = new Button("Goodbye");
  win.add(new Label("Hello"),
          BorderLayout.NORTH);
  Label p = new Label(
   "sin(PI) = "+
    Math.sin(Math.PI));
  win.add(b);
  win.add(p,BorderLayout.SOUTH);
  b.addActionListener(
    new ActionListener(){
      public void actionPerformed
                  (ActionEvent e) {
        win.hide();}});
  win.pack();
  win.show();
  System.out.println("Done");}}
```

The expression `(Listener. (lambda (e) EXPR))` returns an object which implements the 35 Listener interfaces of Java 1.2. The action for all methods of these interfaces is to call the lambda expression on the event.

The Silk code becomes even simpler if a high level GUI library, such as JLIB (written in Silk), is used [TH]. In the example below, the JLIB procedures are written in **bold**

```
;; Silk/JLIB example
(import "java.awt.*")
(import "java.awt.event.*")
(define win (window "Hello"
  (border
    (north (label "Hello"))
    (center (button "Goodbye" (action (lambda(e) (.hide win)))))
    (south (label (string-append "sin(PI) =" (Math.sin Math.PI$)))))))
(.pack win)
(.show win)
(.println System.out$ "Done")
```

SILK Methods

The form `(define-method)` can be used to define a method as part of a generic function. For example, here we define a generic `(iterate)` that iterate over any Java collection, list, or array:

```
(define (identity x) x)

(define (make-iterator more? next seq)
  (lambda (items action)
    (if (more? items)
        (begin (action (next items))
               (iterate (seq items) action)))))

(let ((it (make-iterator .hasMoreElements .nextElement identity)))
  (define-method (iterate (items Enumeration) action)
    (it items action)))

(let ((it (make-iterator .hasNext .next identity)))
  (define-method (iterate (items Iterator) action)
    (it items action)))

(let ((it (make-iterator pair? car cdr)))
  (define-method (iterate (items silk.Pair) action)
    (it items action)))

(define-method (iterate (items Map) action)
  (iterate (.values items) action))

(define-method (iterate (items Collection) action)
  (iterate (.iterator items) action))

(define-method (iterate (items Object[]) action)
  (let loop ((i 0)
             (L (vector-length items)))
    (if (< i L) (begin (action (vector-ref items i)) (loop (+ i 1) L)))))

(define-method (iterate (items Object) action)
  (if (.isArray (.getClass items))
      (let loop ((i 0)
                 (L (java.lang.reflect.Array.getLength items)))
        (if (< i L)
            (begin (action (java.lang.reflectArray.get items i))
                   (loop (+ i 1) L))))
      (error "Don't know how to iterate over " items)))
```

Here we use functional programming style of abstracting the "iterator pattern" into the procedure `(make-iterator)`.

In Java, an array is a subclass of `Object`, so the `Object` version of the `(iterate)` method must check for an array to iterate over its elements.

`(define-method)` is similar to Common Lisp's `(defmethod)` except that

- An argument specializer is a Java class.
- Generic functions take a fixed number of argumens.
- There is no `(call-next-method)`.

Using Java reflection

SILK's implementation makes extensive use of reflection to implement dynamic method invocation, as described below. SILK programs can also make effective use of reflection. For example, a `(describe)` procedure that can describe any object produces output as show in the following dialog requires only 14 lines of code:

```
> (define h (Hashtable. 10))
{}
> (.put h "Fred" 1)
#null
> (.put h "Mary" 2)
#null
> (describe h)
{Mary=2, Fred=1}
 is an instance of java.util.Hashtable
  table: #(#null #null #null #null #null Fred=1 #null #null #null Mary=2)
  count: 2
  threshold: 7
  loadFactor: 0.75
  modCount: 2
  keySet: #null
  entrySet: java.util.Collections$SynchronizedSet@460583
  values: #null
#f
```

```
(define-method (describe (x Object))
  (define (describe-fields x superclass)
    (let ((fs (.getDeclaredFields superclass)))
      (AccessibleObject.setAccessible fs #t) ; Make them all accessible.
      (iterate fs
               (lambda (f)              ; Not static fields.
                 (if (not (Modifier.isStatic (.getModifiers f)))
                     (display (string-append "  " (.getName f) ": "
                                             (.get f x) "\n")))))
      (let ((superclass (.getSuperclass superclass)))
        (if (not (isNull superclass)) (describe-fields x superclass)))))
  (display (string-append x "\n is an instance of " (.getName (.getClass x))
                          "\n"))
  (describe-fields x (.getClass x)))
```

The static method `(AccessibleObject.setAccessible)` is a JDK 1.2 feature that allows an application to control access to reflective information. Here, the code allows accessibility to any field, even private ones. This is essential for the success of `(describe)`.

A simple inspector that extends this idea to use JTables requires about 50 lines of code.

Unique View

Because SILK applications are interactive, Java objects can be manipulated during debugging and software development. This provides a unique view into Java application.

For example, the solution to the following puzzle is easily revealed:

```
/** Can you write a class that can view or alter the secret? **/
public final class Holder {

  private int secret;

  public Holder(int secret) { this.secret = secret;}

  public final boolean isHappy() { return secret > 25; }

  private final Object booster() {
    return new Object() {
        private final void boost() { secret++; }}; }

  private static boolean isHolder(Object x) {
    return x.getClass() == Holder.class; }
}
```

By using reflection to list the methods declared on the class we find something interesting:

```
> (iterate (.getDeclaredMethods Holder.class) print)
static java.lang.Class Holder.class$(java.lang.String)
static int Holder.access$0(Holder)
static void Holder.access$1(Holder,int)
private final java.lang.Object Holder.booster()
public final boolean Holder.isHappy()
private static boolean Holder.isHolder(java.lang.Object)
#f
```

The two methods marked in **bold** are unexpected additions to the Class. They are added by the Java compiler to allow access to the secret by the inner class. This is how inner classes were added to JDK 1.1 without changing the JDK 1.0.2 Virtual Machine. Thus any private field that is referenced by an inner class is available to a Class in the same package.

Limitations

SILK implements all of R^4RS except that:

1. `Continuations` can only be used as escape procedures; that is they can only be called while the surrounding try/catch is still in scope. This is clearly in violation of the language standard.
2. `Strings` are implemented as Java strings and so are not mutable. One can use Java StringBuffers to the same effect. This means that `(string-set!)` is not defined. For many Scheme programs, this should not be a major handicap. For example, in the 23,000 line SLIB library, there are only 42 uses of `(string-set!)`.
3. SILK supports Java’s numeric types and semantics, so for example:

```
> (/ 5 3)
1
```

Implementation

Before discussing implementation details we provide a brief history of the versions of SILK to help motivate the design decisions chosen.

History

SILK versions, up to SILK 1.0, were developed by Peter Norvig [PN]. 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 [4T] test suite which tests Scheme compliance with the R4RS standard. This version also had two procedures `(constructor)` and `(method)` that could be used to construct SILK procedures that invoked a Java constructor or method.

After Peter made this implementation available on the web several people made useful (and sometimes almost identical extensions). This version was a tiny, straightforward Scheme interpreter.

Tim Hickey, who had his own Scheme in Java, adopted SILK, and added JLIB, a library that provides convenient access to the Java AWT. At Brandeis University, JLIB has been in used in an undergraduate/graduate level Computer graphics course (CS155, Spring 1998), and an "Introduction to computers" course (CS2a, Autumn 1997, Autumn 1998) for non computer science majors. Over 1,000 applets have been developed by the students.

The SILK 2.0 version compiled Scheme syntactic expressions into Code objects that could be more efficiently evaluated. This version was started by Peter Norvig and completed by Tim Hickey.

The SILK 3.0 version added generic functions [KA]. The procedure `(import)` would make all the methods of a class accessible as generic functions [AT99]. While this approach provided easy access to Java, it had two problems:

- Since it imported all the methods of a Class rather than just the ones that would be used, some unnecesarry work was always done. This increased startup time.
- Java method names could collide with nongeneric SILK primtives. So these methods were renamed by adding a "#" suffix.

The SILK 4.0 version started as a reimplementation of SILK, by Peter, for use in a book. It had roughly the capabilites of SILK 1.0. This version became the basis of current development. Complete support for Java's numeric types, and a new implementation of generic functions, that is language independant were added.

Object type hierarchy

An important issue for a Java implementaiton of Scheme is, how the Scheme type hierachy should be implemented as Java classes. A reasonable object oriented way is to implement the Scheme types as their own hierachy under Object, as is done by Scheme package [SH] and HotScheme [HS]. This way, the 150 or so Scheme primitives can be implemented as instance methods. While this simplifies the implementation, it makes crossing the Scheme/Java frontier more costly since each Scheme type must be converted in either direction.

The first implementions of SILK implemented a Scheme type with its "closest" Java type. For example, the Scheme types `(char?)` and `(string?)` were implemented as a Java types `Character` and `char[]`, respectively. This allowed Scheme semantics that strings are mutable, while in Java they are not. Also, since a Java String is immutable, it was converted to a Scheme symbol (also immutable) when a Java method returned it to Scheme.

This lead to several complications;

- Any string returned from Java to SILK was interned as a symbol. This lead to excessive memory use in some applications.
- If you actually wanted to get a string from Java you had to do extra work like `(symbol-string (.toString x))`.
- Since a `char[]` is not used that often in Java, it was converted to a `String` when passed to Java, which added overhead. It also made dynamic method lookup more ambiguous.

The current solution to this problem is to implement the Scheme (string?) type as the Java class `String`. This requires Scheme's strings to be immutable, so that `(string-set!)` no longer works. This should not be a big problem since `(string-set!)` is only used in 42 of 23,000 lines of the slib library [SLIB].

Numeric Types

Originally, Scheme's `(number?)` type was implemented by Java classes `Integer` and `Double`. This was reasonable for most Scheme programs. However, Java would occasionally return a `Number` of a different type. After that, SILK arithmetic might or might not work. For example, to compare the last modified date of two files you had to carefully convert a `Long` into a `Double` by first converting it into a `String`:

```
(define (last-modified file)
  (Double. (.toString (.lastModified file))))
```

The current solution to this problem is to implement all Java numeric types. SILK syntax adapts the Java syntax for numbers so that `Long` and `Float` constants can be specified as `3L` or `3.14F`.

Character constants

Scheme and Java have different print representations for character constants. For example, the character "c" is prepresented as #\c in Scheme and 'c' in Java. In SILK we accomodate Java by allowing both #\c and #'c' syntax.

Null

Java, as in C and C++, can return the object `null` to represent the lack of a return value of any particular type. While this is similar to Common Lisp's notion of `nil`, there is no equivalent in Scheme.

The first versions of SILK equated Java `null` with the Scheme empty pair, `'()`. The current version of SILK prints Java null as "#null" which is distinct from "'()". This means that to check for a `null` being returned from Java into the variable `x` one must say:

```
(eq? x #null)
```

rather than

```
(null? x)
```

Primitive procedures

A procedure object must be written for most Scheme primitive procedures (about 150). One way to do this use a separate class, or an anonymous inner class. However, even a tiny inner class produces about a 500 byte class file which can make the resulting .jar file fairly large. Our approach is to use a single class, silk.Primitive, that implements the primitives. Instances of the class are distinguished by an `int` that is used in a case statement to dispatch to the write primitive code. Scheme automatically generates this class.

Dynamic method invocation

A key aspect of SILK and Skij [MT] is that Java methods can be invoked dynamically at runtime. This is done by following the Java method selection semantics. However, while Java does part of the selection at compile time, and the rest at runtime, SILK does the full selection at runtime, based on the runtime types of the arguments.

There are some complications to this approach [MT00] [GHR] [AT], such as:

- A null argument provides no type information. Though this does not seem to happen often in practice.
- Automatic type conversion, such as type widening makes method selection more complicated.

SILK's current approach to not perform any type widening during method selection. This keeps method lookup unambiguous.

Also, since Java `Object[]` arrays are mutable, any method of the reflection API that returns an `Object[]` should be memoized to reduce consing.

Our approach to dynamic method invocation is language independant. SILK determines at read time if a generic call site refers to an instance or static method invocation. In the static case, the lookup is done based first on the method name, and then on the types of the arguments. In the instance case, the lookup is done based first on the target argument type, then the name of the method, and finally the types of the remaining arguments. Over 80% of the cases, the search leads to a single method that is invoked directly [AT99]. Otherwise each potential method is searched by comparing the types of its arguments, to determing the most specific method to be invoked.

The Silk Scheme compiler

Normally, SILK is used on a read-eval-print loop that first reads a SILK expression, then analyzes it intc an efficient form for execution and then executes it:

```
(loop (print (execute (analyze (read)))))
```

The compiler described in this section compile SILK files into Java class files.

The Silk Scheme compiler handles the complete Silk Scheme language (and in particular it fully implements the tail recursion optimization and it self-compiles). The compiler is still rather primitive and currently attempts no optimization except for compiling away reflection, and so the speed of the compiled code is about that of the interpreted code, except for program that make heavy use of Java literals, in which case the compiler roughly doubles the speed of the code.

The fact that the naive compiler does not generate faster code is actually a testament to the efficiency of the interpreter. Indeed, for simple list manipulation programs, like the following naive reverse program:

```
   (define (nrev L)
      (if (null? L) ()
          (append (nrev (rest L)) (list (first L)))))
```

the interpreted code is about half the speed of the corresponding hand coded Java program. On the other hand, for programs that make extensive use of scalar operations, such as the simple countdown procedure:

```
   (define (countdown n)
      (if (> n 0) (countdown (- n 1)) 'done))
```

The Silk code runs about 100 times slower than the corresponding Java program implemented as a recursive procedure (the latter runs out of stack space quickly however).

The current compilation strategy is quite simple. First, the Scheme program is translated into a subset of Scheme containing only constants, java literals, variables, and the following expressions

```
  (set! f v)
  (if A B C)
  (F A1 ... An)
  (lambda R B)
  (quote L)
```

Each file gets compiled to a java class C of the same name. The class has the following members:

```
constructor:
  new C((int)c,(Frame)f) -- creates a closure with with code indexed by c
            and local environment f

instance variables:
  (int)c -- specifies the code of the closure the object represents
  (Frame)f -- specifies the local environment of the closure

instance methods:
  (Object)_Li((Pair)Args) method to evaluate the ith lambda procedure

static variables
  (Object)C.g   globally defined scheme variable g
  (Object)C._Ci ith quoted term
  (Object)C._Li ith lambda
  (Object)C._Ji ith java primitive

static methods:
  C.load() -- evaluates all expressions in the program and loads the
              global defs into the global scheme environment

  C.main((String[])args) -- calls C.load() and then calls the scheme
              procedure (main args), if it has been defined

  (Object)_Ji((Pair)Args) method to evaluate the ith Java literal
```

All reflection is compiled away and replaced by explicit tests for the signature which best matches the call. The compiler uses the silk runtime classes and does no other optimization. Tail Recursion optimization is implemented using the class LCO. An instance of the class stores a pair consisting of a closure and a list it will be applied to. The class has an "eval" static method which tests whether its argument is an LCO object, and applies the LCO closure to the LCO pair, iteratively until a non-LCO object is returned.

The next version of the silk compiler will attempt to analyze the number and types of the arguments to the scheme procedures so that arguments of arithmetic type can be implemented as scalar variables rather than using the Object wrappers, and arithmetic operations can be implemented using the Java operators. Also, tail recursive calls will be replaced by loops. A similar approach has been used in the implementation of

MLj (Benton, et. al.) although in the case of ML, the program is already strongly typed where as a Scheme compiler will need to infer all types.

Another extension we are considering is extending the syntax to allow for scheme specification of Java classes. This could be compiled by extracting the scheme code in the bodies of the methods, constructors, and initializers and compiling it as an inner class with a local namespace. The Java class members would then just invoke the appropriate methods of the inner class and translate the result into the appropriate type (e.g. Integer to int). An example of this extended java syntax is below:

```
(define-class (public Demo (extends Object) (implements Runnable Procedure))
  (field (public Object a) #null)
  (field (private static long b) 5L)
  (field (volatile double c) 4.0)
  (field (protected Object d) #null)

  (static
     (define (fib n) (if (< n 3) 1 (+ (fib (- n 1)) (fib (- n 2)))))
      )

  (constructor (public Demo ((Object x) (byte b)))
     (this)
     (.a$ this x)
     (set! Demo.b b)
     (.c$ this (.doubleValue (Byte. b))))

 (method (public boolean equals ((Object x)))
   (eq? a x))
 (method (public static boolean equals ((Demo a) (Demo b)))
   (eq? a b))
 (method (public static int fib((int n)))
   (fib n))
 (method (private void show ())
   (.println System.out$ (.toString (list ’Demo a b c d))))
)
```

Performance

We have only preliminary preforance results, from a simple microbenchmark that measures the essence of iteration in SILK. This is, the time it takes to compute 100,000 tail recursive iterations.The benchmark is:

```
(define (f n) (if (= n 0) f (f (- n 1))))
(time (f 100000) 1)
```

While there are many Scheme implementations to comapre to, we compare SILK to Guile because both are relatively small Scheme implementations, but SILK is implemented in Java, and Guile is implemented in C.

Here are some results, unfortunately from several different operating system and hardware types:

KC/I	SEC	MHZ	OS	Scheme	Java
456.85	9.14	500	LINUX	SILK	Sun JDK1.3
320.35	6.41	500	LINUX	SILK	IBM JDK1.3
89.48	3.36	266	NT	SILK	Hotspot 1.3beta client
55.99	5.09	110	Sun	Guile	
35.25	0.70	500	LINUX	Guile	

SEC is the time in seconds for (f 100000).

KC/I is the number of kilo machine cycles per iteration, estimated by (* SEC MHZ 0.1). KI/C is an attempt to compare theres number accross different platforms, which should be suspect for different architectures, like Sun vs Pentium.

Note that the IBM JVM did quite well, but the Sun JDK1.3 client Hotspot (beta) does even better on a slower machine! For SILK, this is a consing benchmark, so a good GC will make a difference. Also, a lot of static methods can be inlined. This may explain Hotspot's advantage.

35,000 instructions per iteration may seem like a lot, but neither SILK nor Guile produce code as good as the best Scheme compilers. In the benchmark, n, 0 and f are objects, each iteration has a call to = and f and must convert tail recursion into iteration. Each of these operations, of courese, invoke more functions in C or Java.

Related Work

There are several other Scheme implementations in Java we are aware of, that we briefly describe. The following exhibit shows statistics from these implementations.

Scheme implementation statistics

Implementation	Java files	Lines	Scheme files	Lines	Generics
Silk 1.0	12	1905	0	0	No
Silk 2.0	20	2778	0	0	No
Silk 3.0	28	3508	5	510	Yes
Silk 4.0	24	3869	4	710	Yes
Skij [8]	27	2523	44	2844	Yes
Jaja [9]	66	5760	?	?	No
Kawa [10]	273	16629	14	708	No
Scheme package	252	13,249	39	1,345	No
HotScheme	114	5,674	0	0	No

SILK: The first four rows of the table show sizes for each version of SILK. In the current version, about 1,000 lines of Java, in four classes, are automatically generated from Scheme. This includes about 150 Scheme primitives and the generic Listener class.

Skij: Skij is a Scheme advertised as a scripting extension for Java [MT][MT00]. It is similar in capabilities to SILK and has 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.

(new) (invoke) and (invoke-static) invoke the appropriate Java method using runtime looked up based on all of its arguments. This approach is similar to SILK's. However, SILK's generic functions also allow Scheme methods to be added.

Jaja: Jaja [QC] is a Scheme based on the Christian Queinnec's wonderful book "Lisp in Small Pieces". It includes a Scheme to Java complier 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 U in SILK) to provide globals and utility functions. Unlike Silk, in Jaja, each Scheme type has one or more Java classes defined for it. Similar to SILK, in Jaja, the empty list ‘() is represented as an instance of the class EmptyList. All Jaja objects are serializable.

Kawa: Kawa [PB] 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.

Scheme package: In the Scheme package[SH] the Scheme object Types are arranged in its own class hierarchy. Each primitive procedure is implemented as its own class. This requires over 250 class files, resulting in a 250 kilobyte .jar file.

HotScheme: HotScheme [HS] also uses a Scheme object hierarchy and one class per primitive as the Scheme package does. The hierachy is split so that all Scheme objects, except #f (false) inherit from the TRep (representation of true) class. Primitives come with their own documentation strings. The HotScheme home page is an interactive Scheme applet.

Conclusion

There are many Scheme implementations. This is partly because it is relatively easy to implement a reasonably efficient implementation, and techniques to build a high performance implementation are well understood. It is also partly because, while Scheme is a useful language in its own right, it has found an important role as a scripting language embedded in an application. Guile and STK are two examples of embeddable Schemes implemented in C [Guile][STK].

Beckman argues that scripting languages are inevitable [BB]. In the 80's Jon Bentley popularized the idea of Little Languages [JB]. Such a language can be used to describe in a compact way, one aspect of your project, graphical layout for example. Beckman argues that this decoupling of aspects is essential, because the only other option is to keep changing all the source code. He also argues that little languages often grow to become more complete languages, adding control structure, classes, etc. TCL and Visual Basic are unfortunate examples of this trend. Beckman further argues that Scheme is an excellent choice for a little language, becasue it is also a complete language in which other extension languages can be easily embedded.

SILK is a fertile soil for such minilanguages. As an example, consider an HTML generation minilanguage based on two procedures `(tag)` and `(tag-seq)`. Both return a string. The function `(directory-listing)` generates an HTML page where each file in the directory is represented as a row in a table:

```
(define (directory-listing file)
  ;; Directory listing of file.
  (define (yesify x) (if x "yes" "no"))
  (define (row f)
    (tag-seq (tag ’tr
                  (tag ’td
                       (let ((x (.getName f)))
                         (if (.isDirectory f)
                             (tag ‘(a (href ,(.toURL f))) x)
                             x)))
                  (tag ’td (.length f))
                  (tag ’td (Date. (.lastModified f)))
                  (tag ’td (yesify (.canRead f)))
                  (tag ’td (yesify (.canWrite f))))
             "\n"))
  (tag-seq
   (tag ’head (tag ’title "Directory"))
   (tag ’body
        (tag ‘(table (border 1))
             (tag ‘caption (tag ’em "Directory listing of " file))
             (tag ’tr (map (lambda (x) (tag ’th x))
                           ’(Name Length "Last Modified" Readable Writeable)))
             (map* row (.listFiles file))
             ))))
```

This definition has the same shape as the page it is defining. Namely a head, followed by a body consisting of a table. The table has a caption, and its first row is a sequence of a headers followed by one row per file. It would be hard to get such a compact representation using either a Java Servlet, or Java Server Pgages.

Acknowledgements

We thank Tom Mitchell and Jeff Tustin for reading an early draft, and Geoffrey Knauth for providing some performance results.

References

[4T] Aubrey Jaffer, r4rstest.scm, ftp://ftp-swiss.ai.mit.edu/pub/scm/r4rstest.scm.

[AT99] K.R. Anderson, T.J. Hickey "Reflecting Java into Scheme", Lecture Notes In Computer Science v1616, Pg154-174, 1999.

[BB] Brian Beckman, A scheme for little languages in interactive graphics, Software Practice and Experience, 21, 2, p. 187-208, Feb, 1991.

[CQ] Christian Queinnec, JaJa: Scheme in Java, http://www-spi.lip6.fr/~queinnec/WWW/Jaja.html

[GHR] Suchitra Gupta, Jeff Hartkopf, and Suresh Ramaswamy, Covarient specialization in Java, Journal of Object Oriented Research, p. 16-20, May 2000.

[Guile] http://www.gnu.org/software/guile.

[HS] HotScheme, http://www.stgtech.com/HotScheme/

[JB] J.L. Bentley, More Programming Pearls, Addison-Wesley, Reading, MA, 1988.

[KA]Ken Anderson, A version of SILK with generic functions, http://openmap.bbn.com/~kanderso/silk/jlib/.

[MT] Mike Travers, Skij, IBM alphaWorks archive, http://www.alphaworks.ibm.com/formula/Skij

[MT00] Michael Travers, Java Q & A, Dr. Dobb's Journal, Jan., 2000, p. 103-112.

[PAIP] Peter Norvig, "Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp", Morgan Kaufman, 1992, http://www.norvig.com/paip.html.

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

[PN] Peter Norvig, SILK: Scheme in Fifty K, http://www.norvig.com/SILK.html

[RT] Robert Tolksdorf,Programming Languages for the Java Virtual Machine, http://grunge.cs.tu-berlin.de/~tolk/vmlanguages.html for the

[SH] Stephane Hillion, The scheme package, http://www-sop.inria.fr/koala/shillion/sp/index.html.

[STK] http://kaolin.unice.fr/STk/

[TH] Tim Hickey, JLIB: A Declarative GUI-building library for SILK, http://www.cs.brandeis.edu/~tim/Packages/jlib/jlib.html