llava - Java in Lisp Syntax

Harold Carr
carr@llava.org

ABSTRACT

llava is Java in Lisp syntax (rather than Lisp or Scheme written in Java). llava does not contain special syntax or functions to call Java methods nor does it define an orthogonal set of types (e.g., Scheme strings, Common Lisp arrays). Instead, llava is Java expressed in prefix list syntax with all data being instances of Java classes. llava adds additional types (e.g., Pair, Procedure, Symbol and Syntax) to enable one to work with lists and to define procedures and macros. The paper describes llava language design and implementation, focusing on its transparent reflective invocation system and its package/import system.

Keywords

Lisp, Scheme, Java, Reflection, Modules

1. OVERVIEW

This paper discusses the motivation for Java, presents how it differs from similar efforts, and describes its implementation: the Reflective Invocation system that is the core of enabling llava to provide a prefix notation for Java method calls; how llava procedures are integrated with Java such that method invocations on Java objects take precedence over llava procedures; extending package and import to work with both Java class files and llava procedures and files; and, finally, the llava compilation strategy and "engine"-based runtime.

1.1 Motivation

I once built Lisp systems [1, 2]. Then funding caused me to do C++ [3]. Then Java arrived and programming felt fun again. It reminded me I wanted more. So I used Kawa [4] for interactivity and abstractions (i.e., macros). But I kept running into missing Scheme library items. So I made foreign-function calls to Java. However, I had to convert between Kawa/Scheme I/O, strings, characters, etc., and Java types. (Note: this was with an early version of Kawa.) This lack of interoperability made me realize I didn't want Lisp/Scheme in Java.

That's when I got the idea for llava. Why not create a version of Java that uses Lisp syntax to write Java classes? Then extend that system to include a few special forms (e.g., lambda, set!, if, quote, defmacro), pairs and symbols, a REPL and support for incremental (re)definition.

llava *is* access to Java via Lisp syntax and features. The philosophy of llava is: maximum leverage of Java—only add what is missing or cannot be done in Java.

1.2 The llava language

llava is a prefix version of Java expressed in list structure. As such, it is *not* a version of Scheme with special notation to access Java (like Jscheme [5]) or a version of Common Lisp with a foreign-function interface to Java (like ArmedBear Common Lisp (ABCL) [6]). Table 1 compares language features between Java, llava and these other languages. The table should make it clear that llava is a more direct prefix/list notation of Java - llava's primary goal.

Jscheme uses "dot" notation [5]. This seems to have one advantage over llava: accessing a static field can be accessed as a variable reference, whereas llava chooses to represent this as a procedure call. Table 2 shows this (along with virtual field access).

llava presents Java directly (e.g., true and false rather than Jscheme's #t and #f). Along these lines, rather than invent new module and condition systems, llava exposes package/import and try/catch/finally in a natural way:

```
(package org.llava.demo)

(import java.lang.ArithmeticException)
(import java.lang.Exception)

(let ((bomb 1))
  (define (demo)
    (try
      (if (< bomb 0)
          (throw (new 'Exception "Give up!")))
      (list "Normal result is: " (/ 2 bomb))
     (catch (ArithmeticException e)
       (list "Arithmetic: " e))
     (catch (Exception e)
       (list "Exception: " e))
     (finally
```

Table 1: Notation Comparisons

Java	`llava`	jscheme	abcl
new			
import java.util.Hashtable;	(import java.util.Hashtable)	(import ”java.util.Hashtable”)	
Hashtable ht = new Hashtable();	(set! ht (new ’Hashtable))	(set! ht (Hashtable.))	(setq ht (jnew (jconstructor ”java.util.Hashtable”)))
virtual method calls			
ht.put(”three”, 3);	(put ht ”three” 3)	(.put ht ”three” 3)	(jcall (jmethod ”java.util.Hashtable” ”put” ”java.lang.Object” ”java.lang.Object”) ht ”three” 3)
static method calls			
	(import java.lang.Byte)		
Byte b = Byte.decode(”3”);	(set! b (Byte.decode ”3”))	(set! b (Byte.decode ”3”))	(setq b (jstatic (jmethod ”java.lang.Byte” ”decode” ”java.lang.String”) nil ”3”))

Table 2: More Notation Comparisons

Java	`llava`	jscheme
virtual fields		
import org.omg.CORBA.IntHolder;	(import org.omg.CORBA.IntHolder)	(import ”org.omg.CORBA.IntHolder”)
IntHolder ih = new IntHolder();	(set! ih (new ’IntHolder))	(set! ih (IntHolder.))
ih.value = 3;	(value! ih 3)	(.value$ ih 3)
ih.value;	(value ih)	(.value$ ih)
static fields		
import java.io.File;	(import java.io.File)	(import ”java.io.File”)
File.pathSeparator;	(File.pathSeparator)	File.pathSeparator$

```
        (set! bomb (- bomb 1))))))
```

1.3 Implementation

1.3.1 Reflective invocation

`llava`’s Reflective Invocation (RI) system enables access to Java without special syntax or explicit foreign-function interfaces. Once a Java object is obtained its methods and fields are immediately available:

```
(toLowerCase (substring "Foo-Bar" 4))
  => "bar"
```

`llava` looks up `llava` procedure definitions in internal symbol tables. If a procedure is *not* defined then control goes to the RI. The RI uses Java reflection to invoke the appropriate method by using the procedure call name as the method name, the first argument as the class type (and message recipient) and the types of the remaining arguments.

1.3.2 llava procedures viz Java method calls

A `llava` procedure with the same name as a method of the first argument (and matching argument types) will be invoked if defined with an explicit `lambda` as:

```
(define toLowerCase
  (lambda (x) (toUpperCase x)))

(toLowerCase (substring "Foo-Bar" 4))
  => "BAR"
```

Most of the time that is *not* what you want to happen. `llava` provides an alternate form of `define` that allows Java methods to take precedence:

```
(define (toLowerCase x)
  (- x))

(toLowerCase (substring "Foo-Bar" 4))
  => "bar"
(toLowerCase 3.4)
  => -3.4
```

In other words, if the lookup finds a procedure defined with an explicit `lambda` then that procedure will always be invoked regardless of the argument types. If lookup finds a procedure defined *without* an explicit `lambda` then `llava` will first attempt an RI invocation. If RI finds a method, it will invoke the method and return the results. If RI does not find a method then the `llava` procedure will be invoked. If lookup does not find either type of `llava` procedure then `llava` will throw an exception indicating an undefined top-level variable.

1.3.3 package/import system

Like Java, `llava`’s `import` enables “short”-form access to constructors and static methods and static fields:

```
(import java.lang.Long)
(set! l (new ’Long 34))
(getName (getClass (Long.decode "23")))
```

```
 => "java.lang.Long"
(Long.MAX_VALUE)
 => 9223372036854775807

(import java.lang.System)
(System.out)
 => java.io.PrintStream@1434234
```

When a Java class is imported, `llava` creates a `llava` package for that class and defines procedures in that package to access fields. It also adds the imported package to the list of packages imported by the package doing the importing. Then, during variable lookup, any names with a single dot (".") in them will be expanded to the full package name.

`llava` code may define packages:

```
(package org.example)

(define (whatZone x)
  (let* ((tz (getTimeZone x))
         (n  (getDisplayName tz)))
    n))
```

Details of `llava`'s `package` and `import` implementation and how they interact with Java `classpath`, the file system, and the REPL are given in the main body of the paper.

1.3.4 Compiler and runtime system

`llava` is compiled to a few special forms: application, `begin`, `if`, `lambda`, literal, ref, `set!`. Those forms are written as explicit Java classes, each with a `run` method that takes the current lexical environment and an evaluation engine that also has a `run method`. The forms and the engine cooperate such that the same call to `engine.run` is executed in a Java `while` loop when evaluating tail calls. This makes the `llava` interface to Java properly tail-recursive. The engine evaluator is similar to one used in older versions of Jscheme [7].

1.4 Contribution

The main contribution of `llava` is that it is Java in Lisp syntax rather than another language written in Java. It extends Java by providing macros, procedures, lists, and symbols. Its core feature is automatic Reflective Invocation enabling method calls on Java objects without special syntax, declarations or APIs.

Another contribution is the `package/import` algorithm for enabling dynamic hierarchies of namespaces for top-level variables.

The remainder of this paper gives more details on the design of the language, implementation techniques and performance.

2. THE LLAVA LANGUAGE

The examples shown so far have emphasized defining and calling `llava` procedures as well as creating Java objects and calling methods on those objects. `llava`'s automatic RI system enables both types of calls to look the same. This, coupled with support for `try`, `throw`, `synchronized`, etc., enables one to write procedures in a prefix form of Java.

It is also desirable to write interface and class definitions directly in `llava`. Table 3 shows an extended working example of defining and inheriting (abstract) classes in Java and the equivalent `llava` code. The symmetry between both versions derives from `llava`'s goal to be Java in Lisp syntax. Rather than devise a new notation for defining classes, `llava` provides an intuitive "lispy" syntax such that the mental translation required to move between Java or `llava` syntax is reduced.

2.1 llava language design

It may seem that `llava`'s notation for classes is simple. However, the apparent simplicity is the result of numerous small design decisions.

2.1.1 Methods and fields

Referring to `PointBase` in Table 3, note that a choice has been made in representing fields and methods. The field:

```
    protected int x = 0;
```

could have been represented in many ways. The final two candidates were:

```
    (protected x 0)
    (protected int x 0)
```

The first form was initially considered since type declarations are not necessary in `llava`. However, if methods on classes defined in `llava` are to be called from Java then it is useful to have type information.

Similarly, method representation had several candidates:

```
    (public     getX() (return x))
    (public int getX() (return x))
    (public int getX() x)
    (public int (getX) x)
```

The declaration of the return type could be omitted, but like fields, it is useful to have that information when calling from Java. Also, keeping field types and method return types keeps `llava` representation closer to Java, thereby easing the translation burden. Since all forms in `llava` are expressions there is no need for a `return` statement (`llava`'s `call/ep` can be used for the cases where a `return` is needed. `call/ep` provides continuations with dynamic extent [8]). Finally, since `llava` defines procedures using Scheme syntax it was decided to be consistent and include the method name at the head of the list of arguments. This also makes it easier to distinguish between fields and methods when parsing class defined in `llava`.

The previous sentence points out conflicting constraints in the design of `llava`: stay close to a prefix version of Java while borrowing features from Lisp (i.e., Scheme). To settle

Table 3: Defining Classes

org/llava/pb/PointBase.java

```
package org.llava.pb;

import java.util.LinkedList;
import java.util.List;

public abstract class PointBase {
    protected List history = new LinkedList();
    protected int x = 0;
    protected int y = 0;
    public int getX() { return x; }
    public int getY() { return y; }
    protected void move(int dx, int dy) {
        x += dx; y += dy; moved();
    }
    protected void moved() {
        history.add(this.toString());
        System.out.println("Moved: " + this);
    }
    public List getHistory() { return history; }
    public String toString() {
        return getName() + " x: " + x + " y: " + y;
    }
    protected abstract String getName();
}
```

org/llava/pb/PointBase.lva

```
(package org.llava.pb)

(import java.util.LinkedList)
(import java.util.List)

(public abstract class PointBase
  (protected List history (new 'LinkedList))
  (protected int x 0)
  (protected int y 0)
  (public int (getX) x)
  (public int (getY) y)
  (protected void (move dx dy)
    (+= x dx) (+= y dy) (moved this))

  (protected void (moved)
    (add history (toString this))
    (println (System.out) (+ "Moved: " this))

  (public List (getHistory) history)
  (public String (toString)
    (+ (getName) " x: " x " y: " y))

  (protected abstract String (getName)))
```

org/llava/pb/ColoredPointBase.java

```
package org.llava.cp;

import org.llava.pb.PointBase;

public abstract class ColoredPointBase
    extends PointBase {
    protected String color;
}
```

org/llava/pb/ColoredPointBase.lva

```
(package org.llava.cp)

(import org.llava.pb.PointBase)

(public abstract class ColoredPointBase
  extends PointBase
  (protected String color))
```

org/llava/pb/Point.java

```
package org.llava.p;

import org.llava.pb.PointBase;

public class Point extends PointBase {
    protected String getName() {
        return getClass().getName();
    }
}
```

org/llava/pb/Point.lva

```
(package org.llava.p)

(import org.llava.pb.PointBase)

(public class Point extends PointBase
  (protected String (getName)
    (getName (getClass this))))
```

org/llava/pb/ColoredPoint.java

```
package org.llava.cp;

import org.llava.cp.ColoredPointBase;

public class ColoredPoint
    extends ColoredPointBase {
    public ColoredPoint(String color) {
        this.color = color;
    }
    protected String getName() {
        return getClass().getName()
            + " color: " + color + " ";
    }
}
```

org/llava/pb/ColoredPoint.lva

```
(package org.llava.cp)

(import org.llava.cp.ColoredPointBase)

(public class ColoredPoint
  extends ColoredPointBase
  (public (ColoredPoint (String c))
    (= color c))

  (protected String (getName)
    (+ (getName (getClass this))
       " color: " color " ")))
```

org/llava/pb/ThreeDPoint.java

```
package org.llava.tdp;

import org.llava.p.Point;

public class ThreeDPoint extends Point {
    protected int z = 0;
    public int getZ() { return z; }
    protected void move(int dx, int dy, int dz) {
        z += dz;
        move(dx, dy);
    }
    public String toString() {
        return super.toString() + " z: " + z;
    }
}
```

org/llava/pb/ThreeDPoint.lva

```
(package org.llava.tdp)

(import org.llava.p.Point)

(public class ThreeDPoint extends Point
  (protected int z 0)
  (public int (getZ) z)
  (protected void (move dx dy dz)
    (+= z dz)
    (move this dx dy))

  (public String (toString)
    (+ (toString super) " z: " z)))
```

a design issue it is useful to keep in mind the target user of `llava`. Perhaps a prefix version of Java will hit a sweet spot, resulting in bringing programmers to Lisp, much in the same way Java's similarity with C/C++ enabled C/C++ programmers to more easily migrate to Java. But programmers fluent in Lisp may prefer to have well-known Lisp patterns available. The tension of these conflicting constraints is seen at every step in the design of the `llava` language.

2.1.2 Lexical structure

There are many possibilities in representing Java comments:

```
// single line comment
;  ...

/* block
   comment */
(/* ...)
(-comment- ...)

/**
 * @author me
 */
(/** @author)
(-doc- ...)
```

At present `llava` uses `;`, `-comment-` and `-doc-`. However, `llava` is moving towards settling all language design issues in favor of Java. Therefore support for `//`, `/*` and `/**` seems likely. Using direct Java style block and javadoc comments requires more lexical processing since `read` (and macros) cannot be used to do the work. Using a prefix version of Java style block and javadoc comments seems like a good solution but it is easy to trip `read` up:

```
(/* more . . .)
```

That example will cause `read` to fail looking for the end of a cons cell. Perhaps a compromise to keep the implementation simple while leaning toward Java is to use strings inside of comments:

```
(/* "more . . .")
```

2.1.2.1 Identifiers

When defining procedures and local variables `llava` supports "Lispy" characters:

```
(define *global* 3)
(define (some-procedure 1st-arg 2nd-arg)
  (let ((+sum+ (+ 1st-arg 2nd-arg)))
    (list +sum+ *global* '-foo)))
```

Identifiers for interfaces, classes, fields and methods must follow Java rules.

2.1.2.2 Literals

`llava` reserves Java keywords such as `public`, `abstract` and `static` and adds a number from Lisp such as `define`, `defmacro`, `lambda`, `quote` and `let`. Boolean literals are `true false` rather than `#t #f` or `t nil`.

Even though `llava` tries to settle design decisions in favor of Java there are some cases where Lisp clearly wins out, such as character representation. Java specifies characters as `'a'` but the `quote` reader macro is so fundamental to Lisp history that it cannot be used for characters. In this case `llava` uses the Lisp representation: `#\a`.

2.1.2.3 Operators

Built-in operators are a place that brings the conflicting constraints to the fore:

`foo = bar;`	`(= foo bar)`	`(set! foo bar)`
`x == y`	`(== x y)`	`(eq? x y)`
`x && y`	`(&& x y)`	`(and x y)`
`x != y`	`(!= x y)`	`(not (eq? x y))`
`!x`	`(! x)`	`(not x)`
`x += 1`	`(+= x 1)`	`(set! x (+ x 1))`

At present `llava` uses the Scheme versions. However it seems advisable to change and favor the prefix Java versions, not only to provide an intuitive translation for non-Lisp/Scheme programmers, but to avoid shadowing valid method names such as `and` and `not`.

2.1.3 Types, values and variables

Variables in `llava` interfaces and classes can have modifiers and/or types. Table 3 shows modified and typed fields and return types of methods. `llava` also supports specifying the types of method parameters:

```
(import java.util.Hashtable)
(import java.util.Map)

(public class Table
  (private static int numCreated 0)
  (private Map table)

  (public (Table)
    (+= numCreated 1)
    (= table (new 'Hashtable)))

  (public static int (numCreated)
    numCreated)

  (public void (put (String key) (int value))
    (put table key value))

  (public int (get (String key))
      throws NoSuchElementException
             MinusOneException
    (let ((v (get table key)))
      (cond ((== v null)
             (throw (new 'NoSuchElementException)))
            ((< v 0)
             (throw (new 'MinusOneException)))
            (else
             (intValue v))))))
```

Note that `llava` does not support modifiers or types in `llava` procedure variables and local variables.

2.1.4 Inheritance and declaring exceptions

The previous example has a method `get` declaring checked exceptions. To someone used to Lisp, the declaration seems to be "floating"—to be missing parenthesis. Perhaps it would be better as:

```
  (public int (get (String key))
      (throws NoSuchElementException
              MinusOneException)
    ...)
```

However, for consistency, that would require changing the notation for inheritance in Table 3 to:

```
(public class Point (extends PointBase)
  ...)
```

`llava` currently supports the non-parenthesized version.

2.1.5 *Blocks, statements and expressions*

Local variables and blocks are introduced with `let` and `begin`. As shown in a previous example (Section 1.2), `llava` provides `try/catch/finally`. All `llava` statements are expressions that return values.

2.1.5.1 *Looping*

`llava` supports last-call-optimization for procedures so recursion is encouraged. However, when calling methods on classes defined with `llava`, recursion is discouraged since the method calls go through Java's call stack. `llava`'s `while` macro may be used instead. `do` in `llava` is Lisp's `do` rather than Java's `do/while`.

The only abrupt transfer of control supported by `llava` is `throw` as shown in the `Table` example above. `call/ep` can be used where `break continue` and `return` are needed.

2.1.6 *Anonymous classes*

The syntax for an anonymous Java class is quite cumbersome:

```
AccessController.doPrivileged(
  new PrivilegedAction() {
    public Object run() {
      ...
      return null;
    }
  });
```

At this time `llava` does not support anonymous classes. There is no straightforward prefix notation representation that does not conflict with the notation for procedure application. Also, `llava`'s current compilation strategy makes it hard to separate the anonymous definition from its execution in this context.

3. IMPLEMENTATION

This section shows details of `llava`'s Reflective Invocation system, the `package/import` system and the compiler/runtime system.

3.1 Reflective invocation

`llava`'s Reflective Invocation (RI) system is similar in implementation to Skij's `invoke` [9, 10, 11] and an older Jscheme reflection system [12]. The main difference is in how `llava` uses RI. Instead of an explicit interface, as in Skij's `invoke` or Jscheme's dot-notation, RI is part of `llava`'s variable lookup algorithm.

Consider the code written in `llava` and Skij shown in Table 4. In `llava`, a Java call `o.m(a1, a2)` simply becomes a prefix call `(m o a1 a2)`—no need to explicitly invoke Java. This is accomplished with two techniques:

- `llava`'s undefined identifier handler calls the RI system.
- "generic" procedure definitions that use the RI system.

3.1.1 *Undefined identifier handler*

Referring to Table 4, note that no special syntax or calls are necessary to call `exists` on a `File`. Suppose that a `llava` session does *not* contain a value bound to `exists`. In that case the undefined identifier handler will be executed when `(exists f)` is called. The handler will call the RI to determine if `exists` is a method of the Class type of `f` (although not shown in this example, arguments are also used to find the method). If a method exists then it is invoked and the result returned. If no method is found then `llava` reports an undefined top-level variable.

In the example, `lispList` and `javaList` show that there may be occasions when automatic RI dispatch needs to be avoided. Since `java.io.File` defines a `list` method and if a program wanted to return of list of a file, then calling `list` with one argument, a file, would result in `File.list` being invoked. The example shows the usage of `-list` to avoid this problem.

3.1.2 *Generic procedure definitions*

`llava` supports both Java classes and Scheme-like procedures. To enable the two to coexist, `llava` provides two forms of procedure definitions:

```
(define (floatValue x)
  (list 'floatValue x))

(floatValue 's) => (floatValue s)
(floatValue 10) => 10.0

(define floatValue
  (lambda (x)
    (list 'floatValue x)))

(floatValue 's) => (floatValue s)
(floatValue 10) => (floatValue 10)
```

The first form of `llava` procedure definition is a "generic" procedure definition. In the example, it causes a generic procedure to be bound to `floatValue` in the current package. When `floatValue` is applied, the generic procedure is found in the top-level variable map. The generic procedure mechanism uses the RI for the call. If the RI call succeeds

Table 4: Implicit viz-a-viz explicit RI usage

```
;;; llava                                 ;;; skij

(import java.io.File)
(import java.util.Date)

(define (lastMod name)                    (define (lastMod name)
  (let ((f (new 'File name)))               (let ((f (new 'java.io.File name)))
    (if (exists f)                            (if (invoke f 'exists)
        (new 'Date                                (new 'java.util.Date
             (lastModified f)))))                      (invoke f 'lastModified)))))

(define (sep)                             (define (sep)
  (File.separator))                         (peek-static 'java.io.File 'separator))

(define (lispList name)                   (define (lispList name)
  (-list (new 'File name)))                 (list (new 'java.io.File name)))

(define (javaList name)                   (define (javaList name)
  (list (new 'File name)))                  (invoke (new 'java.io.File name)
                                                    'list))
```

in finding a matching method then the method is executed and the result is returned. If RI does not find a match, then the body of the `llava` procedure is executed instead.

The second form of `llava` procedure definition is a "lambda" procedure definition. In the example, the definition causes a binding of a lambda procedure in the top-level map. When `floatValue` is applied the lambda procedure is found and the body of the `lambda` is executed, regardless of argument types (assuming the correct number of arguments have been supplied).

`llava` uses `lambda` definitions for procedures such as `+` and `!` to avoid unnecessary RI overhead since these identifiers can never be Java method names.

3.2 package/import system

Besides the idea of the `llava` language, another contribution of this work is the implementation of a namespace system to support `package` and `import`. Packages provide partitions for top-level variable bindings.

Each package defines a unique namespace that maps top-level variables defined in that package to values. When accessing a variable while executing in a particular package, say A, that package's map is searched for the variable's value. If found it is returned. If not, the search continues in any packages imported by A, in the order of `imports`. For example, suppose there are three packages:

```
(package a.A)      | (package b.B)     | (package c.C)
(import c.C)       | (import c.C)      | (import b.B)
(import b.B)       |                   | (import a.A)
(define (a x)      | (define (b)       | (define (c)
  (if (eq? x 'c)   |   (cons 'b (c)))  |   (cons 'c (a 'c)))
      'a           |                   |
      (b)))        |                   |
```

In this example, if the current namespace (i.e., package) is `a.A` and `(a 0)` is executed then a call to `(b)` occurs. A binding for `b` is not found in `a.A`'s nor `c.C`'s namespace (in that order). `b`'s binding is found in `b.B`'s namespace. When `b` executes it finds a binding for `c` in `c.C`'s namespace even though control began in `a.A`'s namespace. Similarly for calling `a` from within the `c.C` namespace, resulting in a final value of `(b c . a)`.

The internal representation of this example is shown in Figure 1. The `llava` implementation contains a map from full package names to package objects. Each package object contains a map from identifiers to values bound (in that package) to the variables represented by the identifier. Each package object also contains an ordered "imports" list pointing to package objects representing packages imported by a package.

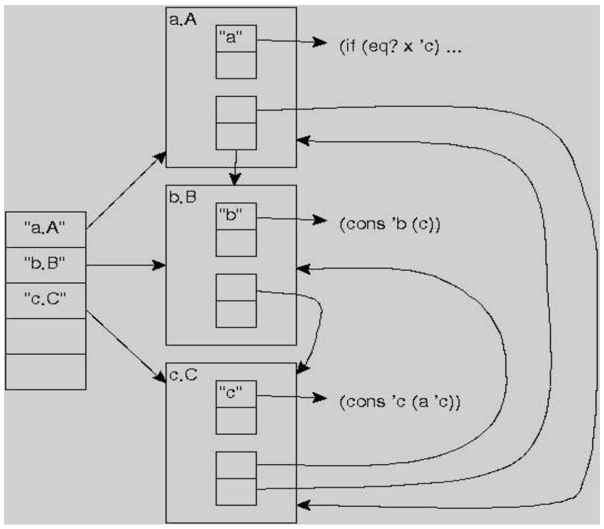

Figure 1: Example package structure

The variable binding search algorithm is straightforward: search for the identifier in the current package's map. If not found look in the environment of imported packages. An optimization is: cache identifier hash codes and use hash tables that take both the hash code and the identifier as arguments. That way the hash code is only computed once per identifier on demand and each hash table uses the hash code rather than recompute it. The identifier is also passed in to resolve hash conflicts within a hash table.

Although the internal package representation and search algorithm are straightforward, loading packages is more complex. Several `llava` features contribute to the complexity:

- Java classes can be imported.
- `llava` provides a REPL, so packages can be defined on-the-fly interactively and then later become files.
- A `llava` package file of a package already represented internally should not be loaded unless it is a new file or has changed since it was last loaded.
- Even if a `llava` package file has not been touched it is still necessary to search the transitive closure of its import list for any package files that may have changed.
- Packages can mutually reference each other so load loops must be detected.
- New packages automatically import `java.lang.*` and `org.llava`.

When `llava` starts up it creates a `org.llava` package object for the built-in `llava` procedures (e.g., `eq?`) as shown in Figure 2. The `org.llava` package object does not contain any package objects in its imports lists. Package objects are also created for `java.lang.*` (the import lists are left empty). A package named `llava-` is created as the initial package for the REPL. The `llava-` package object contains `java.lang.*` and `org.llava` in its imports list in that order. Figure 2 also shows the result of entering `(import a.A)` at the REPL. This causes `a.A` to appear at the head of `llava-`'s import list and causes a package object to be created for `a.A`, `b.B` and `c.C` (the later two not shown).

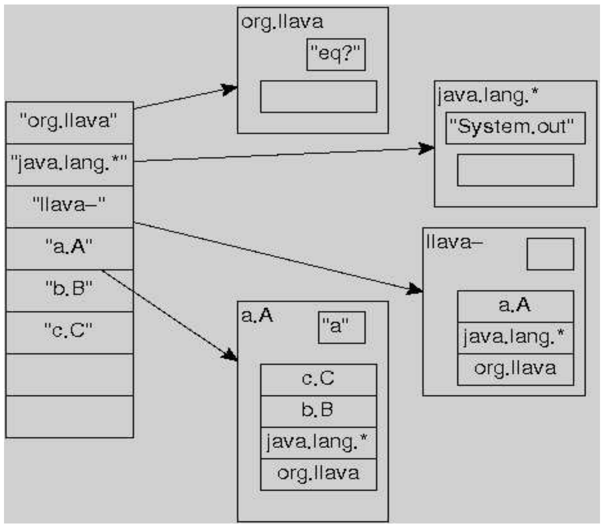

Figure 2: llava package structure at startup

3.2.1 *The import algorithm*

The algorithm for importing `llava` files and Java classes into other packages is shown in Figure 3. The discussion assumes the example packages exist as files. (The discussion begins at the "bottom" of the algorithm and works its way up.)

```
if alreadyImportedInPackage?
    loadLlavaFileIfTouched
else if existsInPackageNameMap
    addToCurrentPackageImportList
    loadLlavaFileIfTouched
else if java class exists
    import class
    classAlreadyImported = true
    addToCurrentPackageImportList
else
    for path in classpath
        if path/file exists
            load file unless being loaded
            addToCurrentPackageImportList
            break
    if file not found in classpath
        throw FileNotFoundException
```

Figure 3: Import algorithm

3.2.2 *Importing llava files*

When `a.A` is imported it is *not* already in the `llava-` package imports list. It is *not* in the full package name map. It is *not* a Java class file in the classpath. Therefore directories and jars on classpath are searched (the final `else` in the algorithm).

`a.A` is found and loaded. When `a.A` is loaded the `package` declaration in its file causes a package object to be created and the current package to be set to `a.A` while stacking the previous package (i.e., `llava-`).

For a moment assume `a.A` does not contain `import` statements. In that case the rest of `a.A` is read. That causes bindings to be entered for top-level variables in the current package that happens to be `a.A`. When loading completes the "package load stack" is popped causing the current package to revert back to `llava-`. Then the last statement of the import algorithm classpath loop is executed causing `a.A` to be pushed onto the front of `llava-`'s import list.

But `a.A` does contain `import` statements. When `(import c.C)` is read while loading `a.A` it causes the import algorithm to be entered recursively. Since `c.C` has not been seen before, all the same steps are taken causing the file for `c.C` to be loaded. `c.C`'s package statement causes `c.C` to become the current package. Then `(import b.B)` causes the same steps to occur until the `(import c.C)` inside `b.B` is reached.

At this point the import algorithm detects that `c.C` is in the process of being loaded so does not load it again. Then the next import algorithm step causes `c.C` to be added to `b.B`'s imports list. One recursive call of the import algorithm terminates, and the previous call continues to load `b.B`, causing a procedure to be bound to variable `b` in `b.B`'s top-level variable map.

When `b.B` finishes loading the next import algorithm step causes `b.B` to be added to `c.C`'s imports list. Loading of `c.C` continues with `c.C`'s next statement `(import a.A)`. The rest of the transitive loading process follows similar steps.

Figure 4 shows the order of loading in this example and the creation of imports list references. Dashed arcs indicate that an imports reference is created but the file is *not* loaded since it is already in the process of being loaded.

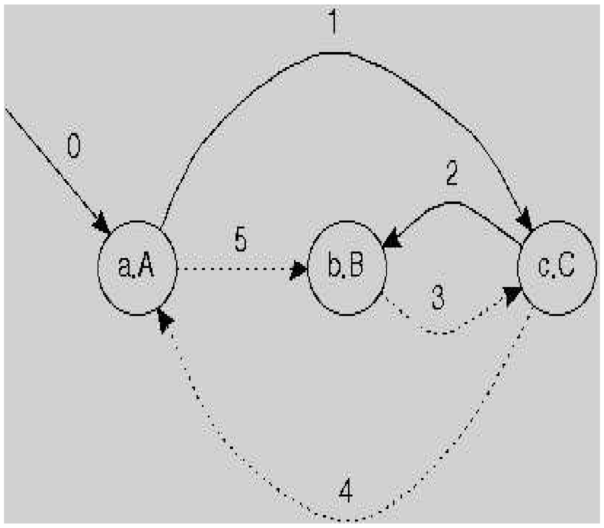

Figure 4: File loading during import

3.2.3 Importing a Java class

`if java class exists`: When a Java class is imported, `llava` creates a package object with variables bound to procedures for public constructors, public static methods and public virtual and static fields in the class. `llava` does not need to define procedures for public virtual methods since they are handled automatically by the RI system. The imports list of the created package is empty. The package importing the Java class has the created package added to its imports list.

3.2.4 Importing existing packages

`if existsInPackageNameMap`: If a package X imports a package Y that is not yet in the X's import list but Y already has a package object, then Y is added to X's imports list. Further, if the file has been touched since it was last loaded then load it again.

`if alreadyImportedInPackage?`: The final (top) part of the import algorithm covers the case where package X imports package Y but Y is already in X's imports list. In this case the only thing that needs to happen is loading the package if it has been touched.

3.2.5 Loading touched files

A package being "touched" can also mean that a package object exists for package X because it was created interactively by typing `(package X)` into the REPL. At some time later, if a file is created for package X and if another package imports X, then the file is loaded.

The algorithm for loading touched files is:

```
if not classAlreadyImported
    loop on classpath
        if lastModified = 0 || fileTime > lastModified
            load file
```

When a package is created interactively its package object contains a `lastModified` field set to 0. When a file is later found for that package then the file is loaded and the `lastModified` field is set to the last modified time of the file. Later, if the package is imported again and the file has been touched (`fileTime > lastModified`) then the file is loaded again.

There are some issues at this point. When reloading a package that has been touched should the previous package object be discarded and a new one created, or should bindings be entered in the existing package? If the former (recreating) then any bindings entered in the REPL are lost. If the later (use existing) then an erasures are not seen. `llava` does recreation.

The loading algorithm shows that `llava` does not support reloading Java classes that have already been imported. It does support reloading packages that contain class definitions written in `llava`. In that case a new `ClassLoader` is used to contain the dynamically generated code (discussed later).

3.2.6 Loading reachable touched files

As discussed, if a package is imported and the file for that package has not been touched, then it is not loaded again. However, the import algorithm (this part is not shown in the pseudo code) loads any touched files in the transitive closure of the non-touched imported file.

3.2.7 Referencing variables by package

As in Java, `llava` allows references to be short or long. For example, `(f)` evaluates to `((c {}) (a {}) (a {}))` (and prints `hello`) for the following code:

```
(package a.A)

(= h
   `(a
     ,(new 'java.util.Hashtable)))
```

```
(package c.C)

(import a.A)
(import
 java.util.Hashtable)
(= h
   `(c
     ,(new 'Hashtable)))
(define (f)
  (list h A.h a.A.h)
  (println (System.out)
           "hello"))
```

In the `llava` implementation, when a variable reference is executed, if the variable's identifier does not contain dots its value is searched starting with the current package then continuing with the imports of that package until found. If the variable contains dots then the package part of the identifier is matched against (partial) package names in the order of the imports list. Note in the example, as in Java, that it is not necessary to import a Java class to create a new instance of that class if the full package and class name is used.

3.3 Compiler and runtime system

The current implementation of llava uses two types of compilation: "Engine"-based for method bodies and procedure bodies, and byte code generation for Class definitions.

3.3.1 Engine-based compilation and execution

Method and procedure bodies are compiled at read time into instances of a Code class. There are specific Code classes for application, application args, assignment, if, lambda, literal, reference, and sequence. At run time these classes interact with an Engine that enables last-call-optimization. The Engine's run method interacts with the run method on Code instances (simplified):

```
public class Engine {
    protected Code code;
    protected ActivationFrame frame;

    public Object run (Code code,
                       ActivationFrame frame) {
        this.code = code;
        this.frame = frame;
        Object result = code.run(frame, this);
        while (result == this) {
            result = this.code.run(this.frame, this);
        }
        return result;
    }
    public Object tailCall (Code code,
                            ActivationFrame frame) {
        this.code = code;
        this.frame = frame;
        return this;
    }
}

public class CodeReference extends Code {
    private int slot;

    public CodeReference (Object source, int slot) {
        super(source);
        this.slot = slot;
    }
    public Object run (ActivationFrame frame,
                       Engine engine) {
        return frame.get(slot);
    }
}

public class CodeIf extends Code {
    protected Code testCode;
    protected Code thenCode;
    protected Code elseCode;

    public CodeIf (Object source, Code testCode,
                   Code thenCode, Code elseCode) {
        super(source);
        this.testCode = testCode;
        this.thenCode = thenCode;
        this.elseCode = elseCode;
    }
    public Object run (ActivationFrame frame,
                       Engine engine) {
        Boolean test =
            (Boolean) engine.run(testCode, frame);
        if (test.booleanValue() == true) {
            return engine.tailCall(thenCode, frame);
        }
        return engine.tailCall(elseCode, frame);
    }
}
```

Code instances can return a value like CodeReference. In that case Engine detects that the returned value is not the Engine itself and Engine.run returns that value. Code instances can also arrange to have the Engine.run method to continue in its loop evaluating further code by calling tailCall. tailCall sets the Code to be executed and the current activation frame in the Engine then returns the Engine itself. Therefore Engine.run will continue its loop. This engine technique was used in older versions of Jscheme [7].

3.3.2 Byte code generation

llava generates byte code for interfaces and classes defined in llava syntax. llava removes method bodies and replaces them with code to call a procedure representing the body as shown in Table 5.

llava does not actually produce a textual representation of the class shell shown in the table. It generates byte codes for that shell. It does create the textual representation of the method bodies (but does not write them to disk).

After generating the class byte code and llava procedures for method bodies, llava loads the procedure code in memory. This causes the correct packages and imports lists to be created for the procedures.

After the procedure packages have been loaded llava loads the byte code into a custom ClassLoader. The static initializer in the class is no longer necessary. It was used at one time when llava wrote the procedure packages to disk.

When calling methods in the generated code, control is transferred to the corresponding llava procedure. This is done by creating a Lisp list of the procedure's identifier followed by the methods (boxed) arguments. This list is then given to F.ce (ce is shorthand for "compile-then-evaluate," F is a static factory).

The evaluation part of F.ce causes the method's corresponding procedure to be invoked. The result of the procedure is the result of F.ce.

4. PERFORMANCE

This section shows the result of executing tak [13] on an Apple iBook 800MHz G3 with 640MB RAM running OS X 10.3.8 using Apple's 1.4.2_05 JRE, build 1.4.2_05-141.4, with the Java HotSpot(TM) Client VM (build 1.4.2-38, mixed mode).

The JRE was started with -XX:CompileThreshold=2 to cause the HotSpot compiler to start early. The tests were run with 100 "warmup" loops to ensure all code is loaded and HotSpot compiled. The times are the average running the test 200 times (after warmup).

Figure 5 shows tak execution times for Kawa, Jscheme and llava. In this test, tak is defined with an explicit lambda to keep llava from entering its RI system. Therefore the measurements are of Scheme procedure calls. Both Jscheme and llava are an order of magnitude slower than Kawa, indicating those systems could benefit from byte code generation of procedure bodies.

Table 5: Compiling classes

```
package org.openhc.llavademo.pb;

import java.util.LinkedList;

import org.llava.F;
import org.llava.Pair;
import org.llava.impl.util.List;

public abstract class PointBase {
    static {
        F.rce("(import org.openhc.llavademo.pb.PointBase-LLAVA)");
    }

    protected java.util.List history = new LinkedList();
    protected int x = 0;
    protected int y = 0;

    public int getX() {
        Pair app = List.list(F.newSymbol("PointBase-LLAVA-getX"), this);
        return((Integer)F.ce(app)).intValue();
    }
    ...
    protected void move(int dx, int dy) {
        Pair app = List.list(F.newSymbol("PointBase-LLAVA-move"), this,
                             new Integer(dx), new Integer(dy));
        F.ce(app);
    }
    protected void moved() {
        Pair app = List.list(F.newSymbol("PointBase-LLAVA-moved"), this);
        F.ce(app);
    }
    ...
}
```

```
(package org.openhc.llavademo.pb.PointBase-LLAVA);

(define PointBase-LLAVA-getX
  (lambda (this)
    (-f 'x this)))

...
(define PointBase-LLAVA-move
  (lambda (this dx dy)
    (-f 'x this (+ (-f 'x this) dx))
    (-f 'y this (+ (-f 'y this) dy))
    (moved this)))

(define PointBase-LLAVA-moved
  (lambda (this)
    (add (-f 'history this) (toString this))
    (println (System.out)
     (+ "Moved: " this))))
...
```

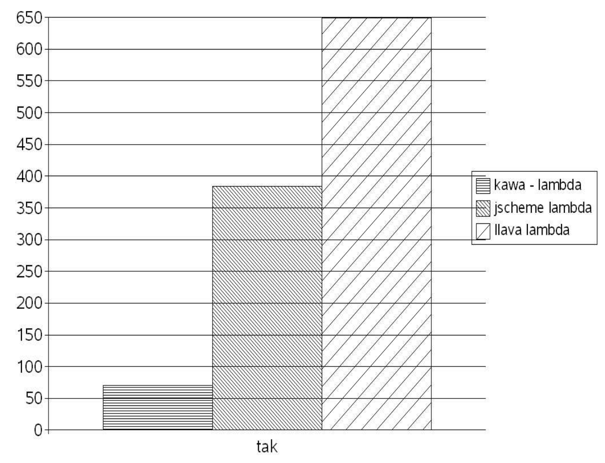

Figure 5: Tak time

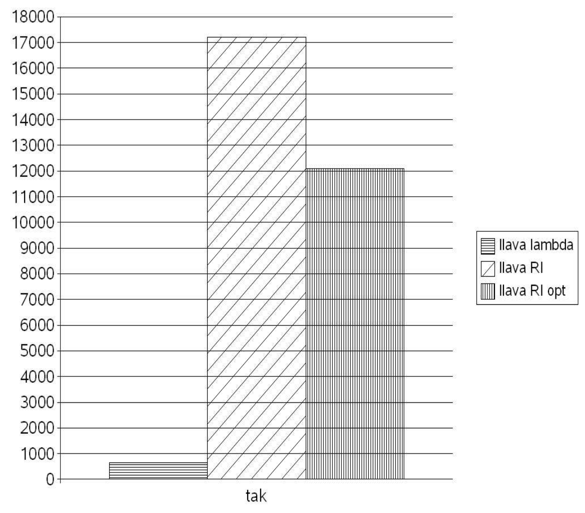

Figure 6: llava tak lambda/RI time

Figure 6 shows the `llava` execution time of `tak` defined with an explicit `lambda` compared to a generic (i.e., RI) definition. This means the comparison is between direct procedure calls and the cost of trying and failing reflection. It is clear that the overhead is large. No time has been spent optimizing any part of `llava` so improvements are possible. In particular, RI indicates failure by throwing exceptions. It would be better to *return* "failure objects." As a quick experiment, parts of RI were rewritten to return failure objects. This resulted in the time shown in the third column labeled "llava RI opt."

Figure 7 show Kawa, Jscheme and `llava` times for the `fprint` and `fread` benchmarks [13]. These times increase our confidence in the accuracy of the `tak` times since it is expected that these times be similar since all three systems implement Scheme `read` and `write` in Java (therefore time spent in the evaluators is minimal).

5. CONCLUSIONS AND FUTURE WORK

`llava` is an idea and an implementation. It is clear that the implementation needs improvement. But, more important at this time, is the idea: Java in Lisp syntax extended with

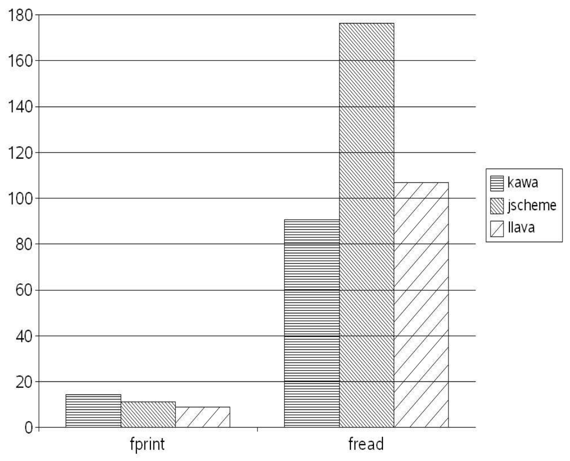

Figure 7: fprint/fread time

procedures and macros. `llava` brings Lisp's adaptability to Java.

The main future work is continued `llava` language design to bring it on par with Java 5. Of course, even if and when Java 5 parity is reached, `llava` language design is unending as Java continues to evolve.

Much implementation work remains. The current implementation of `llava` works on JDK 2.0 and above. The implementation could benefit from Java 5 features and speed. It is clear from the performance figures that the implementation can use extensive optimizations or alternatives.

The RI needs optimization. The first, already mentioned, is to report failure by `return` rather than by `throw`. Better caching of previously found methods could improve performance.

The Engine-based compiler and runtime system could be replaced by byte code generation. Once that is done then `llava`'s current class compilation technique can be replaced with direct compilation of the method bodies inline.

The `llava` implementation has been written such that multiple versions of `llava` can exist independently in the same VM. However, the existing class compilation technique relies on the existence of a "global" static variable to gain access to the `llava` system. This should be avoided.

Despite the implementation's shortcomings, the idea of `llava` is clear: Java in Lisp syntax. `llava` brings the advantages of Lisp to Java in a natural intuitive representation.

6. ACKNOWLEDGMENTS

Thanks to Per Bothner, Michael Travers, Peter Norvig, Ken Anderson and Tim Hickey for their implementations of Scheme in Java. `llava` uses ideas or implementation techniques from their systems. Thanks to Ken Cavanaugh and Tim Hickey for their comments on the final draft of this paper.

7. REFERENCES

[1] R. Kessler, H. Carr, L. Stoller, and M. Swanson. Implementing Concurrent Scheme for the Mayfly distributed parallel processing system. *Lisp and Symbolic Computation*, Vol 5, Issue 1-2 (May 1992) pp 73-93.

[2] P. Pourheidari, R. Kessler, and H. Carr. Moped (a portable debugger) *Lisp and Symbolic Computation*, Vol 3, Issue 1 (January 1990) pp 39-65.

[3] H. Carr, R. Kessler, and M. Swanson. Distributed C++ *ACM SIGPLAN Notices*, Vol 28, Issue 1 (January 1993)

[4] Per Bothner Kawa: Compiling Scheme to Java in LUGM'98: The 40th Anniversary of LISP: Lisp in the Mainstream, Nov. 1998, Berkeley, CA. http://www.gnu.org/software/kawa/

[5] K. Anderson, T .J. Hickey, P. Norvig Silk: A Playful Combination of Scheme and Java *Proceedings of the Workshop on Scheme and Function Programming* , pp 13-22 Rice University, CS Dept. Technical Report 00-368, September 2000. http://www.cs.brandeis.edu/ tim/Papers/Reflection99/ silk2000.pdf

[6] ArmedBear Common Lisp http://armedbear-j.sourceforge.net/

[7] T. J. Hickey, P. Norvig, K. Anderson LISP - a Language for Internet Scripting and Programming in LUGM'98: The 40th Anniversary of LISP: Lisp in the Mainstream, Nov. 1998, Berkeley, CA. http://www.cs.brandeis.edu/ tim/Papers/Reflection99/ lugm.ps

[8] C. Queinnec *Lisp in Small Pieces* Cambridge University Press, 1996

[9] T. Travers Skij http://xenia.media.mit.edu/ mt/skij/

[10] T. Travers Scripting and Dynamic Interaction in Java http://xenia.media.mit.edu/ mt/skij/dynjava/ dynjava.html

[11] T. Travers Dynamic Interaction in Java Dr. Dobb's Journal (25:1) January 2000 http://www.ddj.com/documents/s=889/ddj0001l/ 0001l.htm

[12] K. Anderson, T. J. Hickey, Reflecting Java Through Scheme *Proceedings of the Second International Conference on Metalevel Architectures and Reflection* Springer-Verlag Lecture Notes in Computer Science, vol. 1616, pp. 154-174, 1999. (Reflection'99), Saint-Malo, France, July 19-21,1999

[13] R. P. Gabriel *Performance and Evaluation of Lisp Systems* MIT Press Series in Computer Science, MIT Press, Cambridge, MA, 1985 http://www.dreamsongs.com/NewFiles/Timrep.pdf