llava - Java in Lisp Syntax

Harold Carr
carr@llava.org

ABSTRACT

`llava` is Java in Lisp (lack of) syntax (rather than a 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 (such as Scheme strings or Common Lisp arrays). Instead, `llava` is Java expressed in typical prefix list syntax with all data being native Java data types (e.g., 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.

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 (e.g., lambda) are integrated with Java such that method invocations on Java objects take precedence over `llava` procedures; and extending `package` and `import` to work with both Java class files and `llava` procedures and files; and finally, the `llava` compilation strategy and the "engine"-based runtime.

The rest of this version of this paper is an extended abstract.

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 interactively 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`, `define-syntax`), 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.

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. 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 (perhaps not as "direct" a represent). 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:

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

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

3. IMPLEMENTATION

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. The full paper gives details such as method overload resolution, caching techniques and handling overloaded methods when `null` arguments are given.

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:

```
(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
```

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$`

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 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 it will attempt an RI invocation. If RI does not find a method then `llava` will throw a `LlavaException` indicating an `undefined top-level variable`.

3.3 package/import system

Like Java, `llava`'s `import` enables "short"-form access to constructors and static methods and fields. Unlike Java, `llava import` is required to enable access to virtual 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))
```

The full paper shows how `llava`'s `package` and `import` are implemented and how they interact with Java `classpath`, the file system, and the REPL.

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. More details on the engine-based runtime are given in the full paper. The engine evaluator is similar to one used in older versions of Jscheme.

4. FULL PAPER

The main point 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 Reflective Invocation enabling method calls on Java objects without special syntax.

The full paper gives more details on the features and techniques discussed above and performance results of Reflective Invocation; lookups in the `package import` system; and the Engine-based runtime.

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

[5] K. Andersen, T. Hickey, and P. Norvig jscheme http://jscheme.sourceforge.net/

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