

COMMON LISP FOR JAVA

A New Implementation Intertwined With Java

Jerry Boetje
CS Department
College of Charleston
ILC 2005

1. Abstract

Common Lisp for Java (CLforJava) is an open-source implementation of Common Lisp that executes in the Java Runtime Environment. Under development by undergraduates over multiple semesters, CLforJava differs from other implementations in its meshing of the two languages without a Foreign Function Interface. Following the natural techniques of each language, CLforJava has a Java API to Lisp components and Java methods are accessible via generic functions. The type systems of Lisp and Java are blended together. Documentation from CL forms and doc strings is stored in XML enabling generation of documentation via XSL transformations.

2. Introduction

CLforJava is a new implementation of Common Lisp undertaken by a series of undergraduate CS student teams in the capstone software engineering course. We are now at the end of the 2nd year of development with much accomplished and much remaining. The paper discusses what's unique about this implementation, how it works, what works now, and a projection of the remaining semesters of effort.

3. Intertwining Lisp and Java

3.1. TRANSPARENT INTEROPERABILITY

To be transparently interoperable, both Java and Lisp programmers must feel at home in their respective languages and cultures and yet be comfortable accessing modules, methods, programs, and libraries in the other language. For a Java programmer, it means a Javadoc'd API that defines what can be done with a `Cons` class or the `funcall` methods of a Lisp function. When they compile, they know that the Lisp libraries will be consistent with other Java libraries. For a Lisp programmer, they can access Java objects, classes and methods just as they do with Lisp classes, generic functions, and methods. They require an interactive development environment where there are no special edit-compile-test cycles. The primary com-

ponents enabling interoperability are types, naming, and functions.

3.2. TYPE SYSTEMS

The design of CLforJava relies on some key design patterns for mapping between types.

3.2.1. ROSETTA PATTERN

So named for its importance as a bridge between Lisp and Java, it is the core mapping between equivalent types. The Common Lisp type hierarchy is somewhat tangled (see [Figure 1](#)) and cannot be modeled in the single-inheritance Java class system. However, it can be modeled using only Java interfaces. For example

```
public interface T
public interface Atom extends T
public interface Symbol extends T
public interface Sequence extends T
public interface List extends Sequence
public interface Null extends Symbol, List
```

A Java programmer can determine if an object is a Lisp instance with the phrase `(obj instanceof T)`. All Lisp objects are implementations of a single interface, even though that interface may be a composite, allowing a program to access the Java interface type via reflection. To make the Lisp type name available to Java, we add a static field to the interface whose type is `Symbol`. For example, within the `Null` interface, we add the line

```
public static final Symbol TYPE_NAME =
    (code to create the symbol NULL);
```

Given any Lisp object, a Java program can determine the Lisp type name by accessing its `TYPE_NAME` field.

Making the Java type system available to Lisp is also a simple process. During interface initialization, static code in the interface obtains the Java interface object and adds it to the property list of the Lisp type symbol with a property defined in the `EXTENSIONS` package. A Lisp program merely needs to `get` the property value.

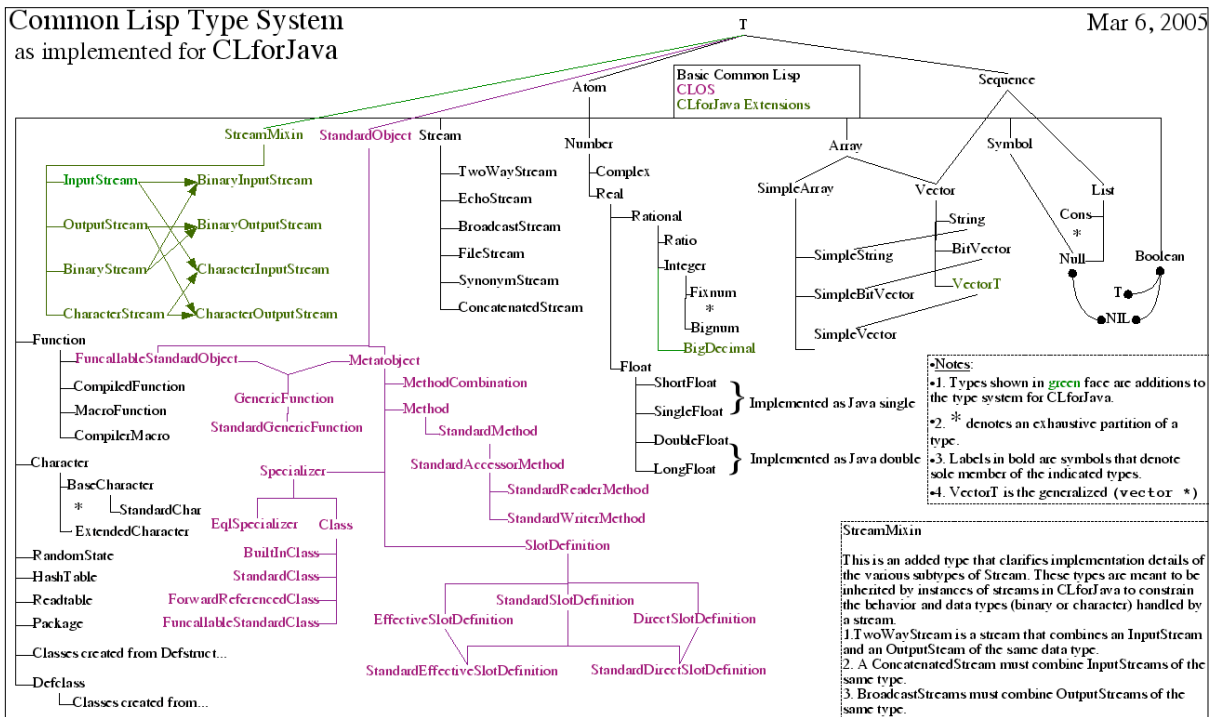


Figure 1 - The CLforJava Type System

3.2.2. TYPE SPECIFIERS

The type-defining interfaces are considered type specifiers and extend one or more of the sub-interfaces of **TypeSpecifier**. **TypeSpecifier** defines an **isTypeOf** method that takes an **Object** and returns a **Boolean**. All of the types shown in [Figure 1](#) extend **AtomicTypeSpecifier**. A type inquiry with an Atomic specifier results in a simple **instanceof** test. Some Atomic types such as **Array** or **Number** also extend **CompoundTypeSpecifier**.

A third class of type specifiers (that do not inherit from **T**) are the compound-only specifications. These types, for example **And** and **Member** (not in Figure 1), extend **CompoundOnlyTypeSpecifier**, a sub-interface of **CompoundTypeSpecifier**. A future version of the compiler will use this information when handling type specifications.

3.2.3. CREATING INSTANCES OF ATOMIC TYPES

Having defined the base types of Common Lisp, we added a factory pattern for creating instances of certain types. In general, non-abstract types such as **List** or **SimpleString** have factories as do some abstract types such as **Integer** or **Character** where the factory chooses the specific type to instantiate.

A factory is a static class nested in the type interface. Each factory class has one or more **newInstance** methods that con-

struct instances of that type or a specialized subtype. For example, the **Integer** interface contains:

```
public static class Factory {
    public static Integer newInstance(int value)
    { ... }
    public static Integer newInstance(String value)
    { ... }
    public static Integer
        newInstance(BigInteger value) { ... }
}
```

Each method returns either a **Fixnum** or a **Bignum** depending on the magnitude of the value.

3.2.4. STRUCTURAL TYPES

Structural types are dynamically constructed via the **defstruct** or **defclass** macros. Both are compiled into Java interfaces that extend **Atom** or **StandardClass** respectively. **Defstruct** definitions that subclass an existing structure type extend that Java interface. **Defclass** definitions that subclass other **CLOS** classes extend the interfaces of all of the listed superclasses, preserving the specified order in the **defclass** form¹.

¹ Java Class objects preserve the order of interfaces specified in the class or interface definition, as does interestingly CLOS preserve the specified superclasses in the same order. A coincidence?

Defstruct forms expand into a static class nested within the interface. This class implements the specified fields for the structure. Its constructor requires all initialization parameters, thus restoring the BOA constructor to its rightful place in the system². The expansion also contains a Factory class for making instances of the structure. The **newInstance** method requires the full set of BOA arguments. If this structure is an extension of an existing structure, the generated class extends the class of the existing class. The **CLOS** types are discussed in [section 7](#).

3.2.5. COMPOUND TYPE SPECIFIERS

Compound type specifiers such as **vector** define constraints on the properties of the general type, e.g. (**vector double-float 100**). Instances of a compound type can be validated by a simple pattern matching algorithm. Subtype relations can be similarly defined. Each of the Java interfaces that defines a compound type contains a pattern that defines a member. Subtypes of these types contain additional pattern information that is merged with the parent pattern to specify the requirements for membership. The root specifier, e.g. **vector**, maintains a list of the compound subtypes.

3.2.6. COMPOUND-ONLY TYPE SPECIFIERS

Unlike the other types, compound-only types are concrete Java classes that perform the expected type checks. Each occasion of a compound-only type specification generates an instance of the appropriate class. For example, an **And** instance is created with a list of types; a **Satisfies** instance is created with a **Function1** object.

3.2.7. SUPPORT FOR JAVA TYPES AND UTILITIES

Several of the core Lisp types also support the common Java idioms for that type. Most notable is the **List** interface, inherited by both **Cons** and **Null** types. The **List** interface extends the `java.util.Collection` interface, giving Java programmers a familiar interface for dealing with a Lisp list. A future version will support the `java.util.List`, `java.util.ListIterator`, and `java.lang.Iterable` interfaces, including support for Java 1.5 generics.

The Lisp number types (sub-interfaces of **Number**) implement the `java.lang.Comparable` interface, enabling Java programmers to compare Lisp numbers directly in Java code. The implementation also supports comparisons between Lisp numbers and their counterparts in the Java library.

3.3. NAMING

Java classes and their public static fields and public methods are accessible via an extension to the Lisp package system. CLforJava extends the **package** type to include a new package type **java-package**. A **java-package** is created in the following situations:

- Evaluation of **find-package** when the supplied name does not reference an existing Lisp package and the value of ***create-java-package*** is not **nil**.

- The Reader encounters a package qualifier that is not an existing Lisp package, the reference is to an exported symbol, and the value of ***create-java-package*** is not **nil**.
- An explicit call to **extensions:make-java-package**.
- In any case, the Java package must be accessible via the Java class loader that loaded the compiled Lisp code.

Java-packages are a subtype of type **Package**, hence support all of the defined methods. However, the methods **intern**, **unintern**, **export**, **unexport**, **packageImport**, **shadow**, and **usePackage** throw exceptions when invoked. **Java-packages** may be used by other packages. Evaluating **in-package** on a **java-package** behaves as expected, but it is not possible to create new symbols in a **java-package** via the Reader. Applying **find-symbol** to a **java-package** will return the appropriate symbol provided it is defined in the Java package. All such symbols are considered exported.

The symbols in a **java-package** have a specific syntax of the form **JavaClassName [.MemberName]** where the **MemberName** (if present) names a static field in the class or a method in the class. In Java, these symbols are a specialized implementation of the **Symbol** interface.

	CLASSNAME	FIELD	METHOD
Value	instance of <code>java.lang.Class</code>	value of the Java field	instance of <code>java.lang.reflect.method</code>
Function	instance of <code>java.lang.reflect.Constructor</code>	--	CLOS method if defined

The **import** function can import individual symbols from a **java-package**.

3.4. FUNCTION ARCHITECTURE

In CLforJava, all Lisp functions are instances of classes that implement the **Function** interface. The only required signature defined is the **apply** method which takes a list of arguments and returns an **Object**. Optionally, a function class may implement one or more of the **FunctionN** interfaces where **N** currently ranges from 0 to 11. Each of these interfaces defines a **funcall** method having **N** arguments. When possible, the compiler will generate calls to the appropriate **funcall** method. In either case, the arguments must be in strict BOA arrangement with no keywords or rest arguments. The compiler arranges to handle the parsing of arguments at runtime. Java coders must provide specific arguments.

3.4.1. GENERAL FUNCTION DEFINITION

All **lambda** expressions are compiled into Java classes implementing at least an **apply** method. The compiler arranges for all of the runtime environment maintenance such as local variable assignment, and closure allocation and creation. By default, the compiler creates a unique name for the Java class and cre-

² CLtL2: pg 483

ates a unique symbol to name the function. An instance of the function class is stored in the symbol's function slot. Users or macro writers can specify either or both via the declarations (see [section 4.2.1](#)).

The Java name should follow the Java language convention specifying package information as a dotted list of identifiers. The name is case sensitive and should be a string. The Lisp name should be a symbol interned in the appropriate package. For example, the Java name of the **CAR** function is `lisp.common.function.Car` while its Lisp name is of course `COMMON-LISP:CAR`.

The compiler arranges to create new instances of the function class as needed during evaluation. At the present time, except for named functions, Java programmers must use Java reflection to create new instances of Lisp-generated function classes. Java programmers can create Lisp functions and pass them to the Lisp system.

3.4.2. NAMED FUNCTIONS

Functions defined by `defun` or `defmacro` are compiled differently from free-standing `lambda` expressions. The compiler produces a named function which is directly available to a Java programmer. Named functions are created as singleton instances of the function class. The function class has a `private` constructor preventing the creation of new instances. The function class contains 2 static final fields. The `SYMBOL` field (type `Symbol`) holds a reference to the symbol naming the function. The `FUNCTION` field holds a reference to the singleton instance of the function class. The type of this field is the function class. By knowing the class name, a Java programmer can invoke the Lisp function directly, for example

```
Cons.FUNCTION.funcall(new Integer(1),
                      new Integer(2))
=> (1 . 2)
```

3.4.3. MULTIPLE VALUES

The JVM is extremely careful about stack management, leaving no room for returning multiple values on the execution stack. At the present time, CLforJava returns a `MultipleValue` object as required. This has the unfortunate side-effect of requiring code to check for a `MultipleValue` return at run time. A later version of the compiler will be smarter in its handling of multiple values.

4. Compilation

4.1. VERSION 1.0 (BOOTSTRAP)

The first version of the compiler was a straight-forward, non-optimizing, classic design using a sequence of a semantic analyzer (SA), intermediate code generator (ICG), and emitter. Both the SA and ICG perform a recursive descent over the form, altering as needed. `let` forms are transformed to `lambda` application. All data types are objects, and all actions translate to `apply` method calls. Special forms have individual, hand-crafted transforms. Approximately half of the CL special forms are implemented. All variable lookups happen dynamically at runtime.

For Java class file generation, we turned to the Oolong JVM assembler. It handles the intricacies of creating a Java classfile (no small feat) from the compiled lambda expressions. These classes are loaded by a custom, in-memory class loader and an instance created by reflection. A non-obvious advantage of Oolong is its textual output that can be printed for debugging code generation. Use of Oolong will likely remain a key component of CLforJava.

4.2. VERSION 2

The next generation of compiler is nearing completion. Its goals are to correct some of the runtime binding mechanisms, improve performance, handle uninterned symbols properly, get better control over Java and Lisp naming, parse the set of ordinary lambda lists, start the process of handling documentation strings, and lay the ground work for transitioning to a Lisp-based compiler.

4.2.1. DECLARATIONS

In addition to the standard, CLforJava defines several declarations that facilitate the Lisp/Java interface:

- `compiler:lisp-name` - a symbol whose function slot will hold an instance of the function class. By default the compiler generates a uniquely-named symbol. The function-generating macros such as `defun` create this declaration to name the resulting lambda expression. This symbol is accessible in Java as a static field - `SYMBOL` - in the implementing java class. The field's type is `Symbol`.
- `compiler:java-class-name` - a string that defines the fully qualified class name of the compiled function. For example, the Java class name of the `car` function is `lisp.common.function.Car`.
- `compiler:singleton` - If present in a lambda expression, the resulting Java class has a private constructor, blocking external creation of function instances. The compiler creates a static field - `FUNCTION` - that holds a singleton instance of the function class. The type of the field is the Java class. The function-generating macros such as `defun` create this declaration. Java code may access the function instance through this field.
- `compiler:documentation` - A string that is the programmer-defined documentation that follows the parameter specification. Generally created by the compiler to regularize declaration handling, this string is combined with other information and stored as the documentation of the function. See [section 5](#).

4.2.2. UPDATING BINDING MECHANISM

The new compiler handles variables differently from the V1.0 compiler where all references were an inefficient dynamic lookup by name. In V2.0, unless the compiler determines that a particular binding `may` escape, local variables are allocated to local variable space in the JVM. This applies to `lambda` arguments and `let` bindings. Bindings that may escape are moved to a `Closure` object and looked up by index.

To bind a special variable, the `bind` and `unBind` methods defined in the symbol. The variable itself maintains a binding

stack. The compiler always generates the equivalent of a `try - finally` block to ensure the unwinding of the stack.

4.2.3. LAMBDA LIST PARSING

The parsing component is fully implemented and tested. However, it is one of the few components that was written directly in Lisp before CLforJava had the capabilities to handle the code. The current system now has the ability to load compiled Lisp code, so we expect to integrate this code next semester. For compiled files, we have yet to implement `load-time-value` needed for storing the parse data.

4.3. FILE COMPILATION

The memory-resident compiler is also the core of the file compiler. The algorithm is:

- Open the source file and set the form list to `nil`
- Until `eof`, read a form and check the `eval-when` rules
 - If `:compile-top-level` is set, recursively `eval` the form in the current environment. The usually entails a recursive call to `compile`.
 - If `:load-top-level`, append the form to the list
- Cons `nil` and `lambda` onto the form list.
- Call the memory-resident compiler.
- The compiler returns an array of Java classes. The first one is the compilation of the outer lambda expression. Get the name of this class.
- Write all of the classes to a `jar` file.
- Add to the jar manifest a `Main` attribute whose value is the name of the outer class.

4.3.1. LOADING COMPILED LISP CODE

When the load function is passed a `jar` file, it accesses the `Main` manifest attribute to obtain the name of the primary class. It then loads and initializes the Java class and, via reflection, creates an instance of the wrapper lambda expression. The loader calls its `funcall` method thereby executing all of the code defined for evaluation at load time. The current compiler does not yet support `load-time-value`. However, its implementation will conform to the current load process. Note that this process is not dependent on the CLforJava file compiler. The loader can load any `jar` file where the `Main` attribute names a class implementing the `Function0` Java interface. This is a handy technique for adding custom Java code to CLforJava.

4.4. TRANSITION TO A LISP-BASED COMPILER

Our current second-generation compiler is still written entirely in Java. However, in the transition from the bootstrap compiler, the data structures were changed to combinations of association and property lists making it relatively easy to re-write major portions in Lisp. Having a Lisp-based compiler will make possible the transition to a third-generation compiler that can handle CPS conversion^[4], optimal closure allocation^[3], and tail call optimization.

A Lisp-based compiler brings with it the rhino-in-the-corner issue of continuations and their JVM implementation. The two techniques under consideration are 1) using Java threads, and 2) implementing continuations as `Throwable` objects.

5. System Documentation

5.1. JAVADOC

CLforJava includes full Javadoc for the Java core, extensions, and system production code as well as Javadoc for the JUnit test suite. A future may include Javadoc generated from the Lisp compiler as it compiles Lisp code.

5.2. DOCUMENTATION STRINGS

The standard technique for creating documentation for functions, macros, variables, etc is a static doc string, accessible via the `documentation` function, closely following the declaration of the object. The standard makes no effort to define the contents of the string. Some implementations augment the string with other information such as the number and names of the parameters to a function, the type of a variable, etc. We propose to go further with this augmentation to provide a complete and flexible XML documentation system.

In our proposed system, each Lisp form that currently supports a doc string will have a `DTD` defining the potential doc components and their structure. For example, a function would have tag names such as `Name`, `HomePackage`, `ParameterList`, `Parameter`, `DocString`, `SourceFile`, `FaslFile`, `FunctionsUsed`, `MacrosUsed`, `SpecialVariablesUsed`. Many of these would have appropriate attributes or substructure. For example, `Parameter` might have `Name`, `HomePackage`, `DeclaredType`, `Kind` (`required`, `optional`, `keyword`, `rest`, `aux` along with appropriate or assumed defaults). The `DocString` component is allowed to have embedded tags for external references and formatting preferences.

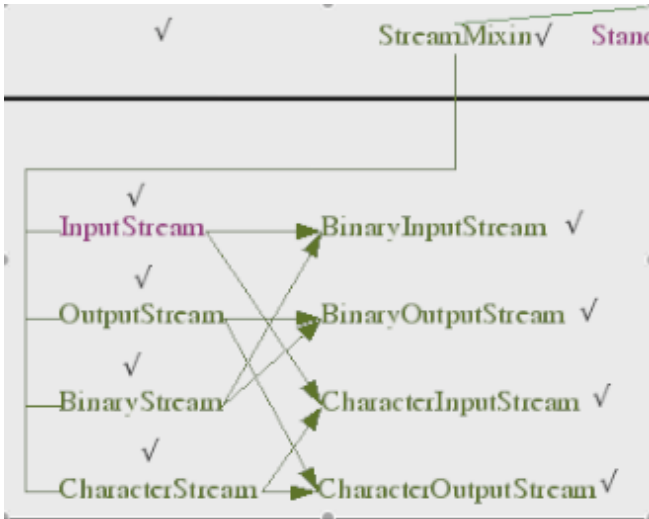
The compiler has the primary responsibility for generating the extra information. The in-memory compiler will generate the XML content and tie it to the compiled object. The file compiler gathers all of the XML content and stores it as a resource in the Jar file containing the compiled classes. For each object with a doc component, the file compiler stores a reference to the XML entry in the compiled object.

Generation of documentation is performed by an appropriate XSLT form. This affords extreme flexibility to the implementer or the user. A transform to simple text blocks could be used for a command-line system. A GUI could generate well-formatted doc including hyperlinks to other documents. A PDF transform could produce printable documentation. Output suitable for DocBook is another possibility.

6. File System

The Common Lisp file system divides the world into rather monolithic character and binary streams. Direction is determined by the `open` function, and invalid operations are flagged only at runtime. The Java file system is not so lax, providing a multitude of classes that specialize the file operations at com-

pile time. CLforJava uses the Java type system to break the file system attributes into a set of mixin interface components that are combined to create the appropriate type of file (see Figure 2). This mixin approach enables a type-safe Lisp file for a Java programmer. For example, a file class that implements `CharacterFileInputStream` does just that.



But creating a type-safe stream violates the operation of Lisp functions that check at runtime. The solution is to define another Java interface that defines all of the possible method signatures and create an abstract system class that implements them. The implementation of each method throws a runtime exception. This class is sub-classed by the concrete classes which independently implement the appropriate Java interfaces, forcing an implementation in that class and effectively overriding some of the superclass methods. An instance of that class can be handed to any file function, but an attempt to invoke a forbidden method (eg, `write-byte` on a `CharacterFileInputStream`) will throw an exception. No need to add runtime tests in the individual Lisp functions.

Character streams read and write objects of type `lisp.common.type.Character` (Unicode characters) which are translated using the stream encoding specification. Character encodings are defined by instances of the `ENCODING` type. CLforJava supports all of the IANA-defined encodings. Instances are created by the `make-encoding` function and can be passed to the `open` function. Since they are types, the Java programmer has a `Factory.newInstance()` method to create new encodings. To the Java programmer, a Character stream acts as a `java.io.Reader` or `java.io.Writer`. File encoding is specified via the `:encoding` option to `open` and defaults to the system default.

7. CLOS

The CLOS design builds on the existing patterns for types and factories. The overall approach is to build core components of the MOP in Java and use that core to implement the CLOS functionality. The basic CLOS metaobjects are represented by

combinations of Java interfaces, abstract Java classes, and classes nested within the interfaces. All of the CLOS definitional macros (`defclass`, `defmethod`, `defgeneric`, etc.) are compiled to interfaces containing embedded factory classes. The following sections describe our current approach.

7.1. CLASSES

New classes are implemented as Java interfaces that extend all of the interfaces that define the direct CLOS superclasses. Since Java interfaces store their superinterfaces in the same form as CLOS superclasses, the same topological sort algorithm is used to create the CLOS class precedence list but using interfaces. Each interface contains a nested Factory class to create instances of the class. Each also contains a nested Definition class that holds structural information such as the class precedence list.

7.2. GENERIC FUNCTIONS

The abstract class `GenericFunction` contains a method to calculate the discriminating function used to select a method for dispatch. It also holds an `Vector` of methods and an instance of `MethodCombination`. The Java class `StandardGenericFunction` extends `GenericFunction` and uses `StandardMethodCombination` for its method combination. `GenericFunction` implements the `Function` interface, providing the `apply` method. Subclasses add the appropriate `FunctionN` methods.

7.3. METHOD COMBINATION

A method combination object is used by a generic function to manage the processing of a method. CLforJava provides a `MethodCombination` interface and a set of classes that implement the CLOS-specified method combinations. The most commonly used class is `StandardMethodCombination` that handles the management of before, after, around, and primary methods. For the benefit of Java programmers, instances of the standard method combination classes are stored in static fields in the `MethodCombination` interface.

7.4. CALLING JAVA METHODS

Accessing Java methods directly from Lisp requires at least two capabilities. The first is the ability to dynamically recognize a non-Lisp class or object. Since CLforJava carefully types all of its objects as implementing `T`, it is possible for the `defgeneric` and `defmethod` macros to distinguish non-Lisp classes. The second is the ability to access a non-Lisp namespace. This is enabled by the Java package extension (see [section 3.3 Naming](#)). For example, to implement in Lisp the Java construct `System.out.println("Hello World");` we would use the following:

```
(defgeneric java.io:PrintStream.println
  (stream object))
(defmethod java.io:PrintStream.println
  ((java.io:PrintStream.println stream) object)
  (call-next-method))
;; now call home
(java.io:PrintStream.println
  java.lang:System.out "Hello World")
```

```
;; or if we have imported the Java symbols
(println out "Hello World")
```

It is possible to intermix Lisp code in the method definition. For example, if we wanted to count the number of times a program writes to `System.out`, we could add the following method:

```
(defmethod java.io:PrintStream.println
  ((eql java.io:System.out) stream) object)
  (incf *system-out-counter*)
  (call-next-method))
```

Note that the Java class is the first argument to `defgeneric`. While the methods can be specialized on multiple parameters, the first parameter is the target of the Java method. Also note that the body of the most general method must contain a call to `call-next-method`. The compiler translates that expression into the desired Java method call. It is possible to optimize a call to the primary method if the compiler can prove that only the most general method will be invoked and that the body contains only `call-next-method`, allowing it to open-code the Java method call.

A goal is a similar extension to the `defclass` macro that enables a Lisp program to create non-Lisp Java classes. This ability would broaden the extent of the intermixing of the languages. An example would be the creation of callback objects used in Swing applications.

8. Engineering the Implementation

The development process is a rather unusual. Creation of CLforJava is a side-effect of an extended undergraduate software engineering practicum course at the College of Charleston. Shortly after taking over the course, I realized that the common approach of having students execute a group project start to finish did not properly prepare them to a career in a modern software development. The gulf was simply too large. To better simulate the “real world development”, the course was reorganized around a long-term development effort to produce an industrial-grade product using industrial tools and industrial processes. Product categories were rejected where there were already many instances such as application servers, blogs, forums, and content management. It was also important to

reject products where students could find most of the code somewhere on the Net - a common occurrence nowadays. Having prior experience developing Common Lisp and with an extensive Java background, I settled on what became CLforJava.

8.1. THE GENERAL PROCESS

The overall engineering process is a spiral model with each semester a complete turn. A semester’s goals include adding new features and improving the product, the development environment, and the documentation. The first four weeks of a semester are devoted to training the students on the extensive tool set, the architecture of the product, and becoming a team. One of the ways this class differs from the usual is that the students are required to work collectively, sharing information rather than acting individually. Participation as a team member is part of their grade. Early in the process the students are assigned bugs to fix, forcing them to delve into the architecture, tool set, and code before starting new development.

Often there are multiple sub-projects and the class is divided up into small teams each handling the design, implementation, and unit testing of a new feature. When the class is enough, one team is assigned to design and code larger test suites for existing code as well as the new development. A benefit from parallel development is that the students experience the need for a source control system and the uses of branching and integrating. They must also document their designs and produce concise and meaningful weekly status reports.

This class requires significant time and effort on the part of the instructor filling the roles of project leader, architect, product manager, code reviewer, lecturer, grader, and occasional coder.

8.2. THE TOOL SET

A significant educational component is the set of development tools used by the students. Most undergraduates have heard of but never encountered bug systems, complex IDEs, and reporting and documentation management systems. Most CS classes never build anything so complex that they require that power. CLforJava is larger and much more complex, requiring the collective efforts of a large and temporally distributed team. It takes the first few weeks of each term to familiarize the students with the tools and how they interact. The following table describes our tool set.

FUNCTION	TOOL	DESCRIPTION
IDE	Netbeans 4	Open-Source IDE providing compilation, debugging, project organization, JUnit and ANT support. Now supported by Sun. http://www.netbeans.org/
Bug Reporting	Bugzilla	Open source bug tracking system commonly used in OSS. http://www.bugzilla.org/
Source Code Control	Perforce	Robust and flexible SCCS supporting labels, branching, integration, reporting. http://www.perforce.com/

FUNCTION	TOOL	DESCRIPTION
Document Management	TWiki	An OSS Wiki system with myriad plugins to handle action tracking, integration with Bugzilla, and more. http://twiki.org/
Status Reporting, Diary	Moveable Type	A blog system used for status reporting and recoding of information that is time-limited, e.g. announcements, questions, etc. RSS feeds. http://www.moveabletype.org/ .
Build System	ANT	XML-based build system. Flexible and extensible. Used directly by Netbeans. http://ant.apache.org/
Java Compiler, RT	Java 5 (1.5)	Basic Java development and runtime system. from Sun. http://java.sun.com/
Test Bed	JUnit	Unit test framework, extended to run large test suites. Supported directly by Netbeans. Currently in design to include Lisp test suites. http://www.junit.org/
Forum	Simple Machines Forum (SMF)	Flexible and extensible message board system. New addition to the tool box. Used for communication with the user community. http://www.simplemachines.org/

8.3. THE PLAN

The following table shows the overall plan for delivering CLforJava. Like any software plan, it will change as circumstances and understanding change. The table describes three

different types of development schedule. The majority of the development is divided into semesters. Some features are more complex than can be delivered in a single semester. And some will be handled as time and resources allow.

DATE	SINGLE SEMESTER	CONSECUTIVE SEMESTERS	OTHER
F03	Types, symbol, cons, basic functions, primitive package system, basic arithmetic, Reader, REP	Bootstrap compiler	
So4	Complete package system, stream and file system, read macros		
F04	Character type, Unicode, re-work numbers and basic arithmetic, Java 5 upgrade		file compiler, lambda list parsing (lisp)
So5	Build lisp in lisp, re-work compiler bindings, Math routines, refactor Reader, finish Unicode integration and Character streams		Web site
F05	Strings, localized comparisons, cleanup, List functions in Lisp, integrate new compiler	CLOS design and implementation (Master's thesis) XML documentation (bachelor's essay)	Migrate V2 compiler to Lisp
So6	Printer, pretty-print, format		

DATE	SINGLE SEMESTER	CONSECUTIVE SEMESTERS	OTHER
Fo6	Sequence functions, arrays, vectors, complex numbers, ratios		Compiler V3, entirely in Lisp, optimizing
So7	Hashtables, Defstruct		
Fo7	Complex types, deftype		
So8	Exception system		
Fo8	Integrate Loop and Series		

9. Benchmarks

We have run few benchmarks as yet due to insufficient functionality to run the benchmark. Our focus is on developing correct code, then making it faster. However, we have run some basic tests against other Lisp implementations, notably CLisp and LispWorks. Our experience is that we are slower by a factor of 2 to 3. However, in **tak** and **factorial**, we are only about 30% slower. We attribute that to Java's implementation of boxed integer arithmetic and of **BigInteger** calculation. By this time next year, we should support most of the Gabriel benchmarks.

10. Summary

We have embarked on a long-term education experiment whose side-effect is a new Common Lisp implementation. This implementation has produced some interesting techniques for melding Lisp and Java, and more will be discovered in the future. Our hope for CLforJava is that it enable more use of Lisp in commercial and research efforts. Follow our progress at <http://clforjava.cs.cofc.edu/>.

11. Licensing and Distribution

CLforJava is an Open Source product licensed under the GNU Public License (GPL) version 2. While we produce and distribute CLforJava within the Computer Science department, we do not yet make it fully and freely available to the public. At this time, it is only distributed on specific request.

12. Acknowledgments

This product and this paper would not have been possible but for the support of the Computer Science department at CofC. The notion of building an industrial-grade product with sequential sets of undergraduates over a number of years was a little daunting to say the least. I'm grateful for the support from George Pothering (current department chair) and Chris Starr (former chair) that enabled me to embark on this adventure. Special thanks goes to my colleague Paul Buhler who encouraged me turn this idea into reality. And of course, all of the CSCI 462 students that have participated in the past 2 years and without whom this product and process would not exist.

REFERENCES

- (1) Steele, Guy. Common Lisp: the Language, Second Edition. Cambridge, MA, Digital Equipment Corp., 1990
- (2) Engel, Joshua. Programming for the Java Virtual Machine. Reading, MA, Addison-Wesley, 1999
- (3) Shao Z, Appel A. "Efficient and Safe-for-Space Closure Conversion". ACM Transactions on Programming Languages, Vol. 22-1, ACM Press, New York. 2000:129-161
- (4) Steele Guy L, Jr. "Lambda: The Ultimate Declarative". MIT AI Lab. AI Lab Memo AIM-379. November 1976. Steele, Guy L Jr.. "RABBIT: A Compiler for SCHEME". Masters Thesis. MIT AI Lab. AI Lab Technical Report AITR-474. May 1978.
- (5) Kiczales G, des Rivieres J, Bobrow D. The Art of the Metaobject Protocol. Cambridge, MA, The MIT Press, 1991
- (6) Lindholm T, Yellin F. The Java Virtual Machine Specification. Mtn View, CA. Sun Microsystems, Inc. 1997
- (7) Kranz, David. "ORBIT: An Optimizing Compiler for Scheme". PhD thesis, Yale University, February 1988. Department of Computer Science, Research Report 632.

A. Example Lisp Function in Java

```
package lisp.common.function;
import lisp.common.exceptions.WrongNumberOfArgsException;
import lisp.common.type.Boolean;
import lisp.common.type.Character;
import lisp.common.type.Function;
import lisp.common.type.Number;
import lisp.common.type.Package;
import lisp.common.type.Symbol;
import lisp.extensions.type.Function2;
/**
 * Implements the Common Lisp function <CODE>EQL</CODE>. Returns true iff
 * its two arguments are <CODE>eq</CODE>, or whose string representations
 * are equal to each other.
 */
public class Eql extends FunctionBaseClass implements Function2 {
    /** Static field holding the singleton instance of the function */
    public static final Eql FUNCTION = new Eql();
    /** Interns the symbol naming the function */
    public static final Symbol SYMBOL = (Symbol)Package.CommonLisp.intern("EQL").get(0);
    /** Connects the symbol and the function instance */
    static { SYMBOL.setFunction(FUNCTION); }
    /** Creates the singleton instance of Eql */
    private Eql() {
    }
    public Object apply(lisp.common.type.List args) {
        if(args.size() != 2){
            throw new WrongNumberOfArgsException("too few arguments given to EQL:");
        }
        Object first = args.getCar();
        args = (lisp.common.type.List) args.getCdr();
        Object second =args.getCar();
        return funcall(first, second);
    }
    /**
     * Funcall takes two objects and returns true if both are identical
     * @param arg1 Object
     * @param arg2 Object
     * @return Boolean
     */
    public Object funcall(Object arg1, Object arg2) {
        try {
            if (arg1 == arg2) {
                return Boolean.T;
            } else if ((arg1 instanceof Number) && (arg2 instanceof Number) &&
                ((Number)arg1).compareTo((Number)arg2) == 0) {
                return Boolean.T;
            } else if ((arg1 instanceof Character) && (arg2 instanceof Character) &&
                ((Character)arg1).compareTo((Character)arg2) == 0) {
                return Boolean.T;
            } else {
                return Boolean.NIL;
            }
        } catch (ClassCastException cce) {
            return Boolean.NIL;
        }
    }
}
```