The JLambda Language

version 3428*

Ian A. Mason[†], David Porter[‡], Carolyn Talcott[§]

July 21, 2019

*Ben Funnell and Linda Briesemeister suggested improvements to this manual.

[†]SRI International, Menlo Park, California, USA, 94025. iam@csl.sri.com
[‡]david.a.porter@gmail.com
[§]SRI International, Menlo Park, California, USA, 94025. clt@csl.sri.com

Contents

1 Introduction

The Java `g2d` package was designed and built as a visualization tool for formal reasoning systems such as Maude [1], PVS [2], and SAL [3]. It is a core component of the IOP system [4], an infrastructure for allowing formal reasoning tools to interoperate, and to interact with users in a transparent and illucidating graphical fashion.

The `g2d` package consists of two intertwined components:

1. An untyped Scheme-like lexically scoped interpreted language, called `JLambda`, that provides a runtime interface to the Java class library, as well as some specific classes that make up the second component. The language makes very heavy use of Java's built in reflective capabilities. It is designed to be efficient and expressive enough to enable full and faithful use of any built in Java classes.

2. A Java class hierarchy, called the `Glyphish` hierarchy [5], inspired by Joel Bartlett's now depricated `Ezd` package [6], and Java's 2D [7] implementation, that provides `JLambda` with sufficient built in classes to effectively construct, at runtime, any desired interactive graphical object.

The `Glyphish` hierarchy has been designed both with several benchmark applications in mind, and with architectural generality. The hierarchy makes heavy use of `JLambda` language. Closures, in particular, provide a rich language in which to describe control flow, event listeners, and even both `static` and non-`static` methods of dynamically created classes.

This paper serves as a reference guide to the `JLambda` language. In section 2 we provide a detailed overview of the constructs of the `JLambda` language. In section 3 we briefly describe the `Glyphish` hierarchy that makes up the second of the two intertwined components of the `g2d` package. In section 4 we provide commentary on a complete prototypical `JLambda` program. In section 5 we describe the implementation of the interpreter, and related work. In section 6 we describe the process by which methods and constructors are resolved. The language itself, as well as the `Glyphish` hierarchy are freely available [8]. In section 7 we describe some common debugging techniques. Finally in section 8 we attempt to document the way the language has evolved over the years.

2 The `JLambda` Language.

The Scheme like `JLambda` language is a minimalistic, call-by-value, left to right evaluation ordering, lexically scoped language with closures. It has exactly the same underlying primitive data types as Java, and access to all of Java's built in packages and classes, as well as any other Java classes found in the class path.

The syntax of the language is based on the usual Lisp notion of an S-expression. An S-expression is either a string of characters or a `List` of S-expressions. A list is either empty, or else contains at least one element.

2.1 Keywords

Since version 1414 JLambda is parsed using Antlr4. As a consequence we have introduced a set of keywords that are not allowed to be used as variables. They are:

```
null
boolean byte double char float int long short
load isnull isobject quote not throw fetch
- + * / % < > <= >= = == !=
narrow instanceof aget lookup
aset update supdate  setattr
concat  and or  if getattr
seq  do let  define  lambda apply invoke sinvoke
object for array mkarray try  catch
reduce nreduce
tuple mktuple nth setnth
```

The only one that is not case insensitive is null.

2.2 Variables

A string of characters is interpreted as a variable, and is evaluated as follows: First see if it is a static Java field e.g.

```
java.lang.String.class,
java.awt.Color.black,
```

or

The .class field does not show up using reflection, and was added as an add hoc case in version 2178.

```
java.awt.geom.GeneralPath.WIND_EVEN_ODD,
```

if it is, return the current value of that field. Next see if it is bound in the current lexical environment, if it is return its current value. Otherwise look to see if it is bound in the current global environment, if it is return its current value, else throw an unbound variable exception.

2.3 Definitions

Global definitions are made using the define form. We provide the usual two Scheme versions. The more general form simply binds the value of <exp_N> (after evaluating the <exp_i> in left to right order) to the identifier <name> (which is not evaluated):

```
(define <name>  <exp_1> ...  <exp_N>)
```

The more specific function or closure definition version is

```
(define <name> (<param_1> ... <param_N>)
    <exp_1> ...  <exp_N>)
```

with $N \geq 0$, which binds the name `<name>` to the corresponding closure, in the global environment. If two `define` expressions assign to the same variable, the first is lost or overwritten. Thus our `define` is similar to a `set`. Note that the latter is just an abbreviation for a variant of the former form. Namely

```
(define <name>
   (lambda (<param_1> ... <param_N>)
       <exp_1> ...  <exp_N>))
```

We will discuss closures, and lambda expressions shortly, along with the lexical binding form `let`.

2.4 Primitive Data

Primitive data is represented *literally* in `JLambda` by a primitive data expression, which is a list of length two:

```
(<tag>  <exp>)
```

Neither the `<tag>` position nor the `<exp>` is evaluated. The `<tag>` should be the name of one of Java's primitive data types:

```
boolean  byte  double  char  float  int  long  short
```

while `<exp>` should be a Java literal, i.e. a sequence of characters. It is parsed as the appropriate data, for example `(int 10)` will be parsed as the number `10`, a Java `int`, while `(char 'C')` and `(boolean false)` will be parsed as Java's character `C`, and boolean `false`, respectively.

If parsing as such fails, then it is the assigned the default zero value of that Java data type (`false` in the case of a `boolean`), and a warning to standard error is issued.

The usual Java literals can be used within these primitive data expressions, see section 3.10, pages 19–27, of [9]. A representative sample is given here:

```
(char 88)          ; evaluates to X
(char 'X')         ; evaluates to X
(char '\130')      ; evaluates to X
(char '\u0058')    ; evaluates to X
(char '\n')        ; evaluates to a new line
(byte 127)         ; evaluates to 127
(byte 128)         ; evaluates to 0 and an error is reported
(int 123456)       ; evaluates to 123456
(int 123456.7)     ; evaluates to 0 and an error is reported
(float 2.5e+7)     ; evaluates to 25000000
(boolean true)     ; evaluates to true
(boolean 0)        ; evaluates to false
```

It should be pointed out that unadorned strings that consist solely of digits, for example 1234567890, are bonafide variables, and can be bound both lexically and globally. To regard them as numbers they need to be wrapped in a primitive data expression. For example, there is nothing wrong with initializing the global environment via the following sequence of expressions.

```
(define 0 (int 0))
(define 1 (int 1))
...
(define 100 (int 100))
```

2.5 Numeric Operations

The usual arithmetic operations are provided. These work on numbers and chars, *not* booleans. These follow the usual Java contortions, see section 5.6, pages 74–76 of [9]. The expressions are evaluated, they are both widened to ints if they are smaller. If one is bigger than an int they are both widened to that type, then the operation is performed.

```
(- <nexp>)
(- <nexp> <nexp>)
(* <nexp> <nexp>)
(+ <nexp> <nexp>)
(/ <nexp> <nexp>)
(% <nexp> <nexp>)
```

To complete the arithmetic operations we provide a corresponding narrowing operation:

```
(narrow <nexp> <exp>)
```

This performs one of Java's 23 primitive narrowing conversions (See section 5.1.3, page 55, of [9]). The expressions are evaluated in left to right order. The second expression <exp> should evaluate to one of the strings: byte, short, char, int, long, double, or float. Indicating the desired primitive data type. The first expression <nexp> should evaluate to a primitive numeric type (including char). The usual loss of precision occurs. Attempting to specify a conversion that is not a narrowing, e.g. (narrow (int 0) "long") will result in an exception being thrown.

2.6 Boolean Relations

We provide the usual binary relations on numbers:

```
(<  <nexp> <nexp>)
(>  <nexp> <nexp>)
(<= <nexp> <nexp>)
(>= <nexp> <nexp>)
```

Non numeric arguments will cause an exception to be thrown.

Like most versions of Lisp or Scheme, there are two forms of equality. Both forms of equality are total boolean relations defined on all types of data:

```
(= <exp> <exp>)
(== <exp> <exp>)
```

The expressions are evaluated in left to right order. The first form, =, is related to the Scheme or Lisp `equals` predicate, while the second, ==, is more like `eq`.

The = expression then returns `true` if either both values are `null`, or neither are `null` and both are booleans of the same value, or both numeric values and are equal, or the second object satisfies the first object's `equals` method. For example `(= (int 0) (long 0))` is `true`. The == is similar but uses pointer equality in the case that both values are objects.

```
(== "foo" (object ("java.lang.String" "foo")))  ; false
(= "foo" (object ("java.lang.String" "foo")))   ; true
(= (char 'X') (int 88))                         ; true
(= (char 'X') (boolean false))                  ; false
```

These first two examples use the `JLambda` notation for calling Java's contructors. This notation will be explained shortly.

There is also an inequality expression:

```
(!= <exp> <exp>)
```

which is the same as `(not (= <exp> <exp>))`.

2.7 Arrays

Arrays may be constructed and manipulated in a direct manner.

2.7.1 Array Construction

There are two forms of array construction, one corresponds to making an empty array of a particular size, the other corresponds to making an array, and assigning its contents. A *zeroed* array of a particular length is constructed via:

```
(mkarray  <type>  <nexp>)
```

whereas an array is constructed *and assigned* via:

```
(array  <type>   <exp_1> .... <exp_N>)
```

where $N \geq 0$. In both cases <type> is either a primitive data tag or else the full name of a Java class, <type> is not evaluated). In the mkarray case the expression <nexp> should evaluate to an integer expression (or something smaller), and an array of that length is constructed and zeroed, according to the nature of <type> in the usual Java way. In the array case of an array of length N is constructed. There can be zero or more elements <exp_i>. The contents of the constructed array are initialized to be the values of the <exp_i>, evaluated from left to right. Each of the values of the expressions should be able to be considered as an element of the class or type represented by the <tag>. null is allowable in the case that the <tag> names a Java class. Widening is allowable in the case of primitive data, as is upcasting to a interface or a superclass in the case that the <tag> names a Java class. Otherwise an exception will be thrown.

In the following examples a0 will be an array of length 88, that contains that many falses, a1 will name an array of integers of length 3, whereas a2 will name an array of objects of length 1.

```
(define a0 (mkarray boolean (char 'X')))
(define a1 (array int (char 'X') (byte 3) (short 7)))
(define a2 (array java.lang.Object java.lang.System.err))
(define a3 (array [I a1 a1)))
(define a4 (array [Ljava.lang.Object; a2 a2 a2))
```

To make an array of arrays one can use two different notations. Early versions of JLambda required that you use Java's quaint array type naming conventions (see section 20.3.2, page 466, of [9]) where Java's internal name for an array class consists of the name of the element type (B for byte, C for char, D for double, F for float, I for int, J for long, S for short, Z for boolean, and L*classname*; for classes or interfaces) preceeded by one or more [characters, depending on the depth of array nesting. Thus [[[[Z would be the name for an array of arrays of arrays of arrays of booleans.

In order to improve legibility, and maintain balanced parentheses, we also accept names generated by the following two clauses:

1. The name of a class or primitive tag is acceptable.

2. If X is acceptable, then [X] is also acceptable, and represents the class of arrays whose elements are those described by X.

In these above examples a3 is an array of integer arrays, and a4 is an array of object arrays. They could also be declared by:

```
(define a3 (array [int] a1 a1)))
(define a4 (array [java.lang.Object] a2 a2 a2))
```

2.7.2 Array Access

The elements of an array are accessed via:

```
(aget <array> <nexp>)
```

which returns the value of <array>[<nexp>], both <array> and <nexp> are evaluated. For example, in the context of the previous array construction examples, both of these expressions

```
(aget a1 (int 0))
(aget (aget a3 (int 1)) (int 0))
```

evaluate to 88. Whereas

```
(aget a2 (int 0))
(aget (aget a4 (int 1)) (int 0))
```

both evaluate to the same instance of the java.io.PrintStream class associated with the standard error stream.

2.7.3 Array Assignment

Similarly an array may be set via:

```
(aset <array> <nexp> <expV>)
```

which sets <array>[<nexp>] to be the value of <expV>. It also returns the value of <expV>. All three subexpressions are evaluated from left to right. As in the case of array construction the value of the expression <expV> must be an allowable element of the array.

So to continue our running examples, we could modify our integer array, a1, by evaluating the expression

```
(aset a1 (int 0) (char 'Z'))
```

and accessing a1 would reflect this change

```
(aget a1 (int 0))
```

and evaluate to 90.

2.8 Strings

The `JLambda` reader will interpret any string of characters beginning and ending with the character `"` as the corresponding Java string (without the two occurrences of the character `"`), actually `"foo"` is interpreted as `(quote foo)`, which evaluates to the Java String `foo`.

String concatenation is provided by the concatenation form:

```
(concat  <exp> .... <exp>)
```

This form of String concatenation, evaluates each expression and concatenates the result (using the `toString()` method of each object returned, and the usual conversions in the case of primitive data). So a simple example of this is that

```
(concat (char '\t') "A " "short " "sentence.")
```

evaluates to a string that prints as

```
        A short sentence.
```

If an argument to `concat` evaluates to `null` an exception is thrown.

2.9 Arbitrary Objects

2.9.1 Object Construction

Arbitrary Java objects may be constructed using the following form:

```
(object (<exp> <exp_1> ... <exp_N>))
```

where `N` ≥ 0. The arguments are, as usual, evaluated from left to right, the first argument `<exp>` should evaluate to string representing the full name of a Java class. If the so named Java class is not `public` an exception is thrown. The interpreter then attempts to find a constructor for that class with matching arguments, and using that constructor and the remaining arguments, constructs the appropriate object. If no matching constructor is found an exception is thrown. We describe this resolution process in more detail in section 6.

For example we can construct a generic object by evaluating

```
(object ("java.lang.Object"))
```

and a wrapper object of the character `X` via

```
(object ("java.lang.Character" (char 'X')))
```

A slightly more interesting example is a frame

```
(define frame (object ("java.awt.Frame" "A Frame")))
```

that we will futz with shortly.

The `null` object reference is obtained by the special form:

```
(object null)
```

which evaluates to `null`.

2.9.2 Field Access

Static and non-static fields of an object can be accessed via:

```
(lookup <target> <field>)
```

The <target> and <field> positions are evaluated from left to right. The interpreter then looks for a field belonging to the object that the <target> evaluates to. The value of that field is returned. If no corresponding field is found an exception is thrown. If <target> does not evaluate to an object an exception is also thrown.

For example the following expressions both evaluate to

```
(lookup a1 "length")
(lookup a4 "length")
```

to the integer 3.[1]

It is, of course, the runtime type of the value of <target> that is used in determining the appropriate field, and its value. (Since this is an interpreted language, there is no sensible notion of compile time type.) Thus `static` fields are probably best looked up via the class rather than an instance if there is the possibility of ambiguity.

The value of a static field may also be accessed from the class via the usual notation

```
java.lang.Math.PI
```

2.9.3 Field Updating

In early versions of `JLambda` we did not provide any support for the setting of fields, either instance or `static`. This was not because it was problematic, but simply because we thought it would be unnecessary, well designed classes enforce object encapsulation of state. If an object's fields are supposed to be updatable, the class will provide methods to do so. Unfortunately, this was overly optimistic, some built in Java classes are no more sophisticated than `C struct`s, for example

```
java.awt.GridBagConstraints
```

Consequently, we now provide means of setting the values of fields, either instance or `static`.

Static and non static fields of an object may be invoked using the following form:

```
(update <target> <field> <exp>)
```

The arguments are evaluated from left to right, and then the interpreter attempts to update the field, whose name is the value of <field> (which should be a `String`), to the value of the expression <exp>. The return value of the entire expression is the new value of the field.

Thus for example we could create two identical points as follows:

[1] That these evaluate correctly is due to an ad hoc clause in our interpreter, since from the point of view of Java's reflection API, `length` is not a field of an array, contradicting the Java language specification, see section 10.7, page 197, of [9]

```
(define p1  (object ("java.awt.Point"  (int 50) (int 50))))
(define p2  (object ("java.awt.Point")))
(update p2 "x" (int 50))
(update p2 "y" (int 50))
```

Alternately, these last two expressions could be abbreviated to:

```
(update p2 "y" (update p2 "x" (int 50)))
```

In the case where one wishes to update a `static` field via the class rather than the object, for example in the situation where there are no instances of the class, we include the following form:

```
(supdate <target> <field> <exp>)
```

In this case `<target>` should evaluate to a string that names the desired class. The evaluation then proceeds in a similar fashion to the non static case. For example, to enable debugging of the entry point to the Graphics 2D actor, one could execute:

```
(supdate "g2d.Main" "DEBUG" (boolean true))
```

2.9.4 Method Invocation

Static and non static methods of an object may be invoked using the following form:

```
(invoke <target> <method> <exp_1> ... <exp_N>)
```

where $\mathtt{N} \geq 0$. The arguments are evaluated from left to right, and then the interpreter attempts to find a method, whose name is the value of `<method>` (which should be a `String`), with the appropriate arguments. If no method is found an exception is thrown. This resolution process is described in more detail in section 6.

Thus for example we could configure and display our `frame` via the following sequence of invocations

```
(invoke frame "setSize"  (int 50) (int 50))
(invoke frame "setLocation" (int 10) (int 10))
(invoke frame "setVisible" (boolean true))
```

which would cause the `frame` to be a square of fifty pixels positioned in the top left hand corner of the screen.

In the case where one wishes to invoke a `static` method via the class rather than the object we include the following form:

```
(sinvoke <target> <method> <exp_1> ... <exp_N>)
```

where $\mathtt{N} \geq 0$. In this case `<target>` should evaluate to a string that names the desired class. The evaluation then proceeds in a similar fashion to the non static case.

So for example if we wished to configure and position our `frame`, so that it was half the size of our screen, and centered therein, we could do the following sequence of instructions.

```
(define toolkit (sinvoke "java.awt.Toolkit" "getDefaultToolkit"))
(define dim (invoke toolkit "getScreenSize"))
(define h (lookup dim "height"))
(define w (lookup dim "width"))
(invoke frame "setSize" (/ w (int 2)) (/ h (int 2)))
(invoke frame "setLocation" (/ w (int 4)) (/ h (int 4)))
(invoke frame "setVisible" (boolean true))
```

To give another example, the class object corresponding to `java.lang.Math` can be obtained, and named, via the expression

```
(define math (sinvoke "java.lang.Class" "forName" "java.lang.Math"))
```

But it would be a mistake to try and access static fields of the `java.lang.Math` class, say `PI`, by then doing

```
(lookup math "PI")
```

because this will actually end up looking for a field called `"PI"` in the class that `math` is an instance of, i.e. `java.lang.Class`. To access static fields via the class use:

```
java.lang.Math.PI
```

2.10 Control Forms

The usual Scheme/Lisp forms are provided.

2.10.1 Lexical Binding

Lexical binding is available via:

```
(let (
       (<var_1> <bexp_1>)
       (<var_2> <bexp_2>)
       ...
       (<var_N> <bexp_N>)
      )
 <exp_1>
 ...
 <exp_M>
 )
```

where $N, M \geq 1$. This expression incrementally augments the current lexical environment. I.e. the binding of `<var_1>` to the value of `<bexp_1>` is visible in the evaluation of `<bexp_2>`, and all subsequent expressions. Once all the bindings have been evaluated, the expressions `<exp_i>` are evaluated in order (in the fully augmented environment). The value of the entire expression is the value of `<exp_M>`. So for example if one were nostalgic for the initial UNIX process file descriptor table, one could construct the following array:

```
(let ((0 java.lang.System.in)
      (1 java.lang.System.out)
      (2 java.lang.System.err))
  (array java.lang.Object 0 1 2))
```

2.10.2 Sequencing

Sequencing is available via the form:

```
(seq <exp_1> ... <exp_N>)
```

where $\mathtt{N} \geq 0$. The value returned from a sequencing expression is the value returned by the last expression in the sequence, or `null` in the case that $\mathtt{N} = 0$.

Thus a simple use of these last two forms would be to construct our centered frame, without littering the global environment with debris.

```
(let ((frame (object ("java.awt.Frame" "A Frame")))
      (toolkit (sinvoke "java.awt.Toolkit" "getDefaultToolkit"))
      (dim (invoke toolkit "getScreenSize"))
      (h (lookup dim "height"))
      (w (lookup dim "width")))
   (invoke frame "setSize" (/ w (int 2)) (/ h (int 2)))
   (invoke frame "setLocation" (/ w (int 4)) (/ h (int 4)))
   (invoke frame "setVisible" (boolean true))
   frame
 )
```

2.10.3 Lambda Expressions and Closures

Closures are obtained by evaluating the corresponding lambda form:

```
(lambda (<param_1> ... <param_N>) <exp_1> ... <exp_M> )
```

where $\mathtt{N} \geq 0$ and $\mathtt{M} \geq 1$. The value of such an expression is a closure, a pair consisting of the lambda expression, and the lexical environment at the time of evaluation. Thunks (the case when `N` is zero) are permissible. As an example of this we could define and name a centered frame factory as follows.

```
(define frameFactory
  (lambda (frameName)
    (let ((frame (object ("java.awt.Frame" frameName)))
          (toolkit (sinvoke "java.awt.Toolkit" "getDefaultToolkit"))
          (dim (invoke toolkit "getScreenSize"))
          (h (lookup dim "height"))
          (w (lookup dim "width")))
      (invoke frame "setSize" (/ w (int 2)) (/ h (int 2)))
      (invoke frame "setLocation" (/ w (int 4)) (/ h (int 4)))
```

```
      (invoke frame "setVisible" (boolean true))
      frame)
   )
 )
```

2.10.4 Closure Application

Application of a closure to the appropriate number of arguments is via the apply form:

```
(apply <exp> <exp_1> ... <exp_N>)
```

where N ≥ 0. To litter our screen with annnoying popups we could then do

```
(apply frameFactory "Viagra popup")
(apply frameFactory "Nigerian bank scam")
(apply frameFactory "Lottery notification")
(apply frameFactory "Random porn site here!")
```

and save other people the time and effort.

2.10.5 Iteration

An iteration form is included based on the Scheme do form [10].

```
(do  (
      (<var_1> <init_1> <step_1>)
      ...
      (<var_N> <init_N> <step_N>)
     )
     (<test> <exp> ...)
     <command> ...)
```

where N ≥ 1, Evaluation of a do form proceeds in two steps, an initialization phase followed by an iteration phase. In the initialization phase the <init_i> expressions are evaluated sequentially in the *outer environment*. Their values are then bound to the variables <init_i> to form the do environment, and the iteration phase commences. In each iteration the following takes place. The <test> expression is evaluated, if it does not evaluate to a boolean an exception is thrown. If it evaluates to true, then the <exp> ... are evaluated in order, and the value of the last expression is the value of the entire do form. Otherwise each of the <command> ... expressions are evaluated for their effect, then the variables in the do environment are updated, *sequentially*, to be the values of the <step_i> forms, and the next iteration commences.

So a simple example is

```
(do (
     (i (int 0) (+ i (int 1)))
     (str " " (concat str " " i))
    )
    ((= i (int 10))  str)
    (invoke java.lang.System.err "println" str)
    )
)
```

prints out a triangle of numbers:

```
 1
 1 2
 1 2 3
 1 2 3 4
 1 2 3 4 5
 1 2 3 4 5 6
 1 2 3 4 5 6 7
 1 2 3 4 5 6 7 8
 1 2 3 4 5 6 7 8 9
```

In Java 1.5 a *for-each* construct was introduced, which enables the programmer to conveniently iterate over a collection or array. `JLambda` supports this construct via the `for` form:

```
(for <var> <rexp> <exp_0> ... <exp_N>)
```

Here, the range expression `<rexp>` should evaluate to either an integer, a string, an array object or an instance of either of the following classes or interfaces.

```
java.util.Collection
java.util.Iterator
java.io.File
g2d.jlambda.Range
```

otherwise an exception is raised.

The last three alternatives were added in `3428` in July 2019.

The sequence of expressions `<exp_0> ... <exp_N>` are then evaluated once for each element of the array/collection/iterator, in an environment in which `<var>` is bound to the current element. In the case of a `File` object the elements being iterated over are the lines in the file, not including the line separator. The value of the entire `for` form is the value returned by the final evaluation of `<exp_N>`.

A `g2d.jlambda.Range` object has three `final` integer fields: `begin`, `end`, and `stride`. When iterating over such an object, the first element is `begin`, and then evey subsequent increment of `stride`, ending before `end`.

So for example

```
(for c (object ("g2d.jlambda.Range" (int 0) (int 10) (int 2)))
     (invoke java.lang.System.out "println" c))
```

prints out

```
0
2
4
6
8
```

In the case of the integer n, or string evaluation behaves as if the `<exp>` argument was the collection whose elements are $0 \ldots n-1$, in the case of an integer or as a char array in the case of a string.

In this simple example,

```
(for x
     (array int (int 3) (int 5) (int 7))
     (invoke java.lang.System.err "println" x))
```

prints the list of numbers:

```
3
5
7
```

whereas

```
(for x (int 8)
    (if (and (> x (int 2)) (!= (% x (int 2)) (int 0)))
       (invoke java.lang.System.err "println" x)
   )
)
```

does much the same thing.

The `reduce` and its generalization `nreduce` were introduced in January 2018. The `reduce` form:

```
(reduce <exp0> <funexp> <exp1>)
```

evaluates each argument from left to right. The first argument `<exp0>` should evaluate to an iterable object in the sense of `for` (that is either an `Collection`, array, or integer). The second argument should evaluate to a closure of arity 2. The third argument is called the `accumulator`. If the iterable object is empty the form returns the `accumulator`. Otherwise it proceeds by apply the closure to the next element in the iteration and the `accumulator`. The value returned by the closure becomes the value of the `accumulator` in the next step of the computation. When all the elements of the first argument have been iterated over, the `accumulator` is returned.

A simple example is a function that given an integer `n` makes an array of length `n` whose `i`th entry is `i`.

The treatment if integers as set theoretic ordinals (i.e the set of ordinals less than them) was introduced in version `2180`, as a simpler version of the mystical `do` construct. In version `2184` we tweaked the parsing so that `body` could be an implicit sequence of expressions. Strings became iterable in February of 2018, because Ian likes (non-whitespace aspects of) Python.

```
(define makeArray (n)
  (let ((arr (mkarray int n))
        (fun (lambda (elem accum) (aset accum elem elem) accum )))
    (reduce n fun arr)))
```

The `nreduce` form:

```
(nreduce <exp0> ... <expN-1> <funexp> <accum>)
```

is analagous, except that it takes `N` iterable arguments, and the closure must then be of arity `N + 1`. Computation proceeds until one of the iterable objects is exhausted, when the current value of the `accumulator` is returned. A simple example is

```
(define squarer (elem0 elem1 accum)
  (let ((s (* elem0 elem0))) (aset accum elem1 s) accum))

(let ((accum (mkarray int (int 10))))
  (nreduce (apply makeArray (int 10)) (int 10) squarer accum)
```

which makes an array of length 10, whose `i`th entry is `i*i`.

2.10.6 Branching

Both binary and ternary forms of the branching primitive are allowed:

```
(if <test> <then>)
(if <test> <then> <else>)
```

In both the binary and ternary forms the `<test>` should return a `boolean`, The binary form returns `null` if `<test>` is false.

2.10.7 Boolean Operations

The related propositional forms are as usual:

```
(and <exp> .... <exp>)
(or <exp> .... <exp>)
(not <exp>)
```

The expressions are evaluated from left to right, their values should be `boolean`, otherwise an exception is thrown. In the case of `and`, evaluation is terminated on the first `false` value, while in the case of `or`, evaluation is terminated upon the first `true` value encountered.

2.10.8 Testing for Nullness and Objectness

As a convenience we provide a simple test to determine if an value is `null`:

```
(isnull <exp>)
```

which returns `true` iff the the expression `exp` evalutes to `null`. Similarly we provide a simple test to determine if an value is neither `null` nor primitive data:

```
(isobject <exp>)
```

which returns `true` iff the expression `exp` evalutes to a value that is an instance of a subclass of `java.lang.Object`. In other words is neither `null` nor primitive data.

These two operations are new in version 1407.

2.10.9 Quoting

Finally evaluation can be prevented via the usual quote form:

```
(quote <exp>)
```

which returns the unevaluated `<exp>` as data, either a `String` or a `List`.

Pretty sure no-one has ever used quoted data, so I may eliminate this for anything other than strings.

2.11 Subtyping

There is a version of Java's `instanceof` operator. The expression:

```
(instanceof <exp> <exp>)
```

returns Java's idea of whether or not the value of the first `<exp>` is an instance of the class named by the (`String`) value of the second `<exp>`. For example, the expression

```
(instanceof "java.lang.String" "java.lang.String")
```

will evaluate to `true`. Note that there is a clear distinction made between primitive data and the associated wrapped form, an expression such as

```
(instanceof (int 7) "java.lang.Integer")
```

will evaluate to `false`.

2.12 File loading and single line comments

One can load a file using the `load` form:

```
(load <exp>)
```

The expression `<exp>` should evaluate to a string naming the file. If this file is a jar file (i.e. ends with `.jar`), then the jar file is loaded. Otherwise the file is assumed to contain jlambda expressions, and the contents of the file, is parsed and then evaluated.

In the file loaded *comments* may appear. The `JLambda` language originally only had single line comments. The comment character is the semicolon `;`, with the proviso that it is preceeded by whitespace, or begins a new line. With the introduction of Antlr4 parsing, we have added both C and C++ style comments.

```
; this is a single line comment

//this is also a single line comment

/* this
on the other
hand, can go on
for as long as it wants, but just like in C++
CANNOT be nested
*/
```

The ability to load a jar file was added in version `2295`. This feature is experimental, and is not designed to replace the common use of the `JLAMBDA_CLASSES` environment variable.

2.13 Class Names, Inner Classes, and Enums

All of Java's classes may be accessed by their full names. There is no import mechanism, other than that provided by the lexical and dynamic binding primitives:

```
(define Vector "java.util.Vector")

(let ((Line "java.awt.geom.Line2D$Double")
      (victor (object (Line (int 0) (int 0) length (int 0))))
  (object (Vector victor)))
```

Note that, following Java's reflection conventions, inner classes are accessed via the `$` rather than the `.`. So for example the elements of `NodeType enum` in `g2d.graph.IOPNode` would be accessed by

```
g2d.graph.IOPNode$NodeType.RULE
```

2.14 Exceptions

One can throw an exception using the `throw` form:

```
(throw <exp>)
```

The expression `<exp>` should evaluate to an instance of `java.lang.Throwable`, otherwise a somewhat different exception will be thrown. There is also a `try catch` form for handling any exceptions that may be generated. It is closely related to the Java form, without the option of multiple catches, or a `finally` clause.

```
(try <exp_1>
     ...
      <exp_N>
     (catch <var> <cexp_1> ...  <cexp_M>))
```

where `N` ≥ 1. Evaluation of this form proceeds by first evaluating the `<exp_i>` in order, if this succeeds without generating an exception, then the value of the entire expression is the value returned by `<exp_N`. However, if evaluation of one of the `<exp_i>` generates an exception, then this exception is bound to the variable `<var>`, and `<cexp_1>` through `<cexp_M>` are evaluated in this larger environment, and the value of the last `<cexp_M>` is the value of the entire `try catch` expression. For example,

```
(try (throw (object ("java.lang.Throwable")))
     (catch var (boolean true)))
(try (boolean false)
     (catch var (boolean true)))
```

the first expression evaluates to the boolean `true`, while the second evaluates to the boolean `false`.

Since version 2022 we have included the class `g2d.jlambda.Result` to make it easy to *throw* an arbitrary piece of data up to and enclosing `try`. One simple use is to return early from a for loop.

```
(let ((digits (array int (int 0) (int 2) (int 3) ... (int 10000)))
      (pi java.lang.Math.PI)
      (ball (sinvoke "g2d.jlambda.Result" "create" pi))
      (result
       (try
        (for i digits (if (= i (int 7)) (throw ball)))
        (int -1)
        (catch result
               (seq
                    (if (instanceof result "g2d.jlambda.Result")
                        (lookup result "value")
                       (int -2)))))))
  result)
```

The class `g2d.jlambda.Result` subclasses `java.lang.Throwable`, and has a *static* factory method `create` for each basic data type, as well as the `java.lang.Object` class. The value used to create the `g2d.jlambda.Result` object can be obtained by looking at the object's `value` field.

2.15 Tuples

Tuples are *experimental*. Tuples are, mixed type, fixed length sequences. They are instances of the `g2d.jlambda.Tuple` class bestowed with some special features endowed by the interpreter. One can store both objects and primitive data in a tuple. Apart from calling the tuple constructors directly, a tuple can be created in one of three ways:

```
(tuple <exp0> ... <expN - 1>)
(mktuple <integer>)
(mktuple <java.util.Collection>)
```

The first creates a tuple of length `N` with the given arguments as its members. The second creates an empty tuple of the desired length, while the third form creates a tuple that has the same members as the collection. The class `g2d.jlambda.Tuple` subclasses `java.util.AbstractCollection` with none of the optional methods implemented. One can recognize a tuple by the primitive

```
(istuple <obj>)
```

which is the same as asking

```
(instanceof <obj> "g2d.jlambda.Tuple")
```

Because tuples are collections, one can iterate over a them using any of the forms `for`, `reduce`, and `nreduce`. You can also determine a tuples length by using the `size` method, but a quicker way is to reference its `final` field:

```
(lookup <tuple> "length")
```

To access an element of a tuple use:

```
(nth <tuple> <integer>)
```

To update a tuple use:

```
(setnth <tuple> <integer> <value>)
```

Tuples were inspired by Seus' carts and python tuples. Their main use was envisaged to be as accumulators for the `reduce` and —`nreduce` forms, and perhaps as an easy way of returning multiple values from a function. *The urge to pass them to Java methods should be resisted, since Java is blind to their mixed type nature, and so the two disparate views of the world may be unhappy.*

3 The `Glyphish` Hierarchy

A description of the `JLambda` language would not be complete without touching on the `Glyphish` hierarchy mentioned in section 1, and in particular it's use of the `JLambda` language. This class hierarchy belongs to the `g2d.glyph` package, whose class structure is shown below:

```
              Identifiable
                   |
              Attributable
                   |
                Glyphish
           /       |        \
   Glyph      GlyphList      ClosureGlyph
```

Javadoc documentation for these and other classes can be found at [11].

3.1 The `Identifiable` Class

Graphical objects that are manipulated by other actors (for example formal reasoning tools such as Maude and PVS) may be assigned unique identifiers by them. Unique identifiers are strings. To access an object by it's unique identifier the fetch form is used:

```
(fetch <exp>)
```

attempts to fetch the object whose unique identifier is the value of the expression `<exp>`.

An object, belonging to a class extending `Identifiable`, may be assigned a unique identifier using the `setUID` method. It is an error, i.e. an exception will be thrown, to assign the same identifier to two objects, and it is also an error to assign an identifier to an object that has already been assigned an identifier. Similarly one accesses an particular object's unique identifier by the `getUID` method. If the instance of the `Identifiable` class has not been assigned a unique identifier the `getUID` method returns `null`. Finally one can unassign a name to an `Identifiable` object by using the `unsetUID` method.

```
(invoke <exp> "setUID" <exp>)
(invoke <exp> "getUID")
(invoke <exp> "unsetUID")
```

In November 2013 we added the specialized `JLambda` forms that offer a more general feature. Unique identifiers can be manipulated directly by the dedicated forms:

```
(setuid <exp> <exp>)
(getuid <exp>)
```

The first form enables one to assign to the value of the first argument the value of the second argument as it's uid. It is slightly more general in that the first object need not be an instance of the `Identifiable` class. Any non-`null` Java value can be assigned a uid, though the uid must still be a string. To unassign a uid to an object, once simply assigns `null` as its new uid. To get the uid of an object, `obj`, one can simply use the expression `(getuid obj)`. So after `(setuid foo "bar")`, for some object `foo`, we would have that `(getuid foo)` returned `"bar"`, and `(fetch (getuid foo))` was just `foo`.

3.2 The `Attributable` Class

An object belonging to a class extending `Attributable` may have new fields added dynamically, in the form of attributes. An attribute has a name and a value. For efficiency attributes are manipulated via the dedicated forms:

```
(setAttr <exp> <exp> <exp>)
(getAttr <exp> <exp> [<default>])
```

rather than by the corresponding methods of the object involved. The `getAttr` form has an optional default argument. This is the value returned, in the case where the object has no attribute by the specified name. If no default is specified, and no attribute is found, then `null` is returned. In both forms the first expression evaluates to the target instance, the second evaluates to the name of the attribute (which should be a string), while in the `setAttr` form the third evaluates to the new desired value, while in the `getAttr`form the optional third expression evaluates to the default value to be returned, in the case where the object has no attribute by that name.

Attribute values are arbitrary objects, including closures. An attribute with a closure value corresponds to a dynamic method. Two ways to make use of such a dynamic method is with an apply expression, or by invoking one of the `Closure`'s many `applyClosure` convenience methods directly on the attribute.

```
(setAttr att "method" (lambda (x) ...))
...
(apply (getAttr att  "method") (int 7))
...
(invoke (getAttr att "method") "applyClosure" (object null))
```

In section 4 we present in detail an example that, amongst other things, uses attributes to implement mouse draggable graphical objects.

3.3 Glyphs

In Joel Bartlett's `Ezd` package a `Glyph` was something that knew how to draw itself, typically as a sequence of shapes, and was capable of accepting and responding to user inputs. The `Ezd` package was developed using the early Java 1.0 event model, and relied on the `java.awt` package as it was then, in Java 1.0.

We adopt a similar, though distinct, conceptual approach. Our approach has been strongly influenced by the newer Java event model, and the clean two dimensional graphics supplied by the Java 2D API. In particular we make heavy use of the `AffineTransform` class of the `geom` subpackage of `java.awt`, and the newer 2D implementations of the `Shape` interface, also of the `java.awt` package. We have also taken advantage of the `Closure` class provided by the `JLambda` language.

In our approach the root class of all things *glyph-like* is the `abstract` class `Glyphish`. There are three related but distinct aspects to the `Glyphish` class:

1. How a `Glyphish` instance depicts or portrays itself graphically, typically this is done at the request of an encompassing `Component`.
2. How a `Glyphish` instance handles input events from the keyboard and mouse, again typically delegated by an encompassing `Component`.
3. How a `Glyphish` instance positions or transforms itself within it's encompassing `Component`.

The abstract `Glyphish` class has three *main* immediate concrete subclasses:

- the `Glyph` class,
- the `GlyphList` class, and
- the `ClosureGlyph` class.

The `Glyph` class is the simplest of the direct subclasses of `Glyphish`. A `Glyph` instance has a single `java.awt.Shape`, border colour, fill colour, and stroke width. A `GlyphList` is a composite, it consists of an ordered list of `Glyphish` things. A `ClosureGlyph` is the most dynamic, it requires `JLambda` closures to implement all the `abstract` methods required by the `Glyphish` API. It provides, in essence, a way of defining, at runtime, `Glyphish` instances whose methods are defined at runtime, rather than compile time.

3.3.1 Depiction

A concrete `Glyphish` instance portrays itself by implementing the abstract method

```
void paint(java.awt.Graphics2D g2d);
```

declared in the `Glyphish` class. In the case of a `Glyph` instance it will draw itself according to its `java.awt.Shape` field, fill colour, border colour and stroke. In the case of a `GlyphList` it merely delegates, in order, to all of the `Glyphish` elements in its list. The `ClosureGlyph` responds by applying it's private

```
Closure paintClosure;
```

field to the appropriate `Graphics2D` object of the `java.awt` package.

3.3.2 Events

`Glyphish` instances are capable of handling any input events, i.e. instances of the `InputEvent` class of the `java.awt.event` package. To determine whether a `Glyphish` instance is the desired target of an `Input` event the abstract method

```
boolean inside(Point2D p);
```

of the `Glyphish` class is used, here the class `Point2D` belongs to the `java.awt.geom` package. The `Glyph` class implements this by using it's `java.awt.Shape`'s field corresponding

```
boolean contains(Point2D p);
```

method. The `GlyphList` instance implements this by iterating through it's list of `Glyphish` elements calling each element's `inside` method, and returns `true` if one returns `true`, else it returns `false`. The `ClosureGlyph` responds by applying it's private

```
Closure insideClosure;
```

field to the appropriate `Point2D` object.

The `Glyphish` class implements each of the `Input` event listener interfaces: `MouseListener`, `MouseMotionListener`, and `KeyListener`, all of the package `java.awt.event`. They do so in a uniform way. For each method in the listener interface a `Glyphish` instance has a `Closure` object associated with it. For example in the case of the

```
void mouseClicked(MouseEvent e);
```

method of the `MouseListener` class, the `Glyphish` class has the private field

```
Closure  mouseClickedAction;
```

This closure will have arity 2, and uses Luca Cardelli's trick of having a self argument to implement Java's `this` pointer. Calling the

```
mouseClicked(MouseEvent e)
```

method would result in the `clickedAction` closure being applied:

```
clickedAction.applyClosure(this, e);
```

where the `this` pointer is of the `Glyphish` instance that is responding to the `MouseEvent` instance `e`.

3.3.3 Transformation

Positioning, moving, and animating `Glyphish` instances is done by applying affine transformations (e.g. translating, rotating, shearing, and scaling) to them.

A concrete instance of the `Glyphish` class transforms itself by providing an implementation to the abstract method

```
void transform(AffineTransform a);
```

of the `Glyphish` class. In the case of a `Glyph` instance, the `AffineTransform` a is applied to the instance's `Shape`. In the case of the `GlyphList` instance, it is applied to each `Glyphish` element of its list. While the `ClosureGlyph` simply applies it's private

```
Closure transformClosure;
```

to the `AffineTransform` object `a`.

3.4 Closures

Another use of the `Closure` class provided by the `JLambda` language is in creating event handlers. For each Java listener class, we provide a corresponding template class that implements the desired listener interface, and extends the `Attributable` class. An instance of the template class can be specified by providing `Closure`s for each of the required methods. As mentioned above, we have adopted the convention that these closures take two arguments, the *self* parameter, followed by the *event* parameter.

A simple example of this, corresponding to Java's `ActionListener` interface of the `java.awt.event` package, is the `ClosureActionListener` class of the `g2d.closure` package. The action listener interface requires the solitary

```
void actionPerformed(ActionEvent e)
```

to be implemented. Consequently the `ClosureActionListener` class has a private field

```
Closure actionPerformedClosure;
```

a setter for this field:

```
public void setActionClosure(Closure closure){
    if(closure.getArity() == 2) {
            actionPerformedClosure = closure;
    } else { // report an error }
}
```

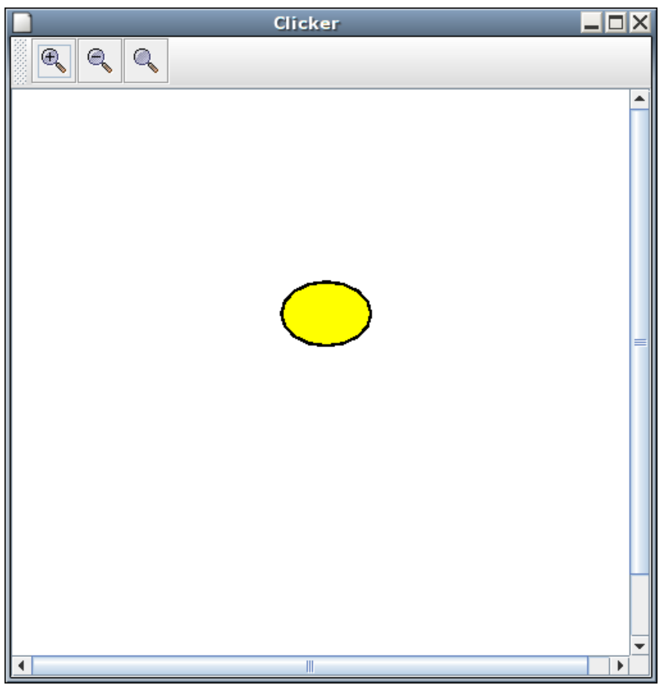
The required `actionPerformed` method is then simply implemented by:

```
public void actionPerformed(ActionEvent e){
   if(actionPerformedClosure != null) {
     actionPerformedClosure.applyClosure(this, e);
   }
}
```

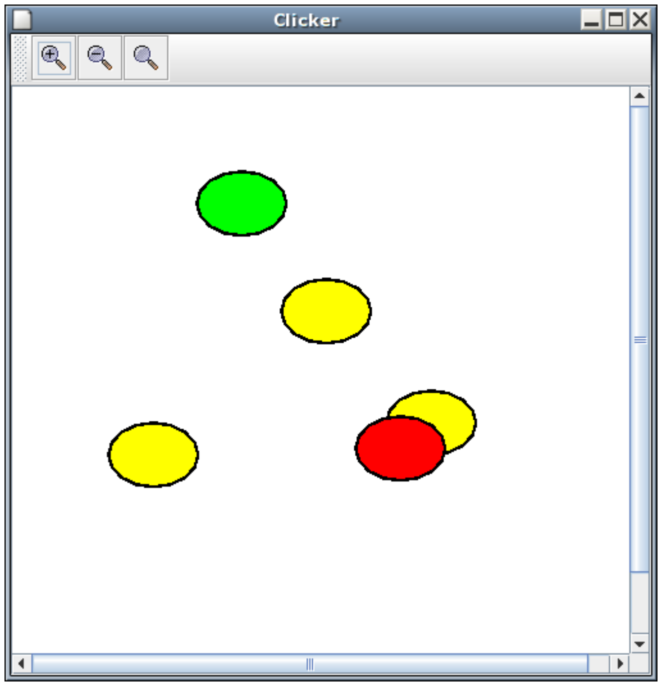
4 A Complete `JLambda` Program

In this section we present a complete example of a `JLambda` program. The program, `clicker`, contains the most interesting features of `JLambda`, namely, the creation and manipulation of Java objects, and the use of closures [12] as event handlers.

The `clicker` program, a full listing of which can be found at [13], constructs graphical objects using the `g2d` class hierarchy [11]. When the `clicker` program is run, it displays a window containing a single circular node. Clicking anywhere in the window creates a new node at that point. The following is a screenshot of the `clicker` program window showing a single node.



The user can change the colour of an existing node by shift-clicking on it; a dialogue box will appear, allowing the user to select a new colour for the node. The following is a screenshot of the `clicker` program window showing several nodes, some of which have had their colour changed by the user.



Finally, existing nodes can be dragged using the mouse.

In Section 4.1 we present an overview of the lexical structure of the clicker program code, in an attempt to orient the subsequent discussion. Section 4.2 contains details each section of the `clicker` program code, along with an explanation of its operation.

4.1 The `clicker`'s Lexical Structure

To aid navigation of the `clicker` source code, we provide the following representation of its lexical structure.

```
(let ((h ...)
      (w ...)
      (black ...)
      (yellow ...)
      (stroke ...)
      (ellipse ...)
      (view ...)
      (frame ...)
      (mkNode
       (lambda (...)
         (let ((node ...)
               (pressed ...)
               (released ...)
               (dragged ...)
               (clicked ...)
```

```
               (trans ...))
           (seq ... (invoke ...) ...))))
      (clickedV ...))
  (seq ...))
```

4.2 Code Listing for **clicker**

The `clicker` program begins by creating the main window object; it defines bindings for some Java primitive types, fields, and objects, which are used to specify the initial properties of the node object, and the `g2d.swing.IOPFrame` object, which is the program's main window.

```
(let ((h (int 50))
      (w (int 70))
      (black java.awt.Color.black)
      (yellow java.awt.Color.yellow)
      (stroke (object ("java.awt.BasicStroke" (float 2.5))))
      (ellipse (object ("java.awt.geom.Ellipse2D$Double"
                        (int 0) (int 0) w  h)))
      (view (object ("g2d.swing.IOPView"
                     (boolean true) (boolean true))))
      (frame (object ("g2d.swing.IOPFrame" "Node Example" view)))
```

Note that the use of these bindings relies on a particular semantics of `JLambda`'s `let` expression: that each binding incrementally augments the lexical environment.

The next code section defines a closure which, when invoked, constructs a node object.

```
(mkNode
 (lambda (xPos yPos)
   (let ((node (let ((temp (object ("g2d.glyph.Glyph"
                                    ellipse black yellow))))
                 (seq (invoke temp "setStroke" stroke)
                      temp)))
```

In creating the node, the closure uses several bindings from the previous section to specify initial properties, such as colour, of the node.

The `mkNode` closure also creates several closures for each of the mouse click events to which the node responds, and binds them to the variables `pressed`, `released`, `dragged`, and `clicked`. Later in the program, these closures will be registered as event handlers for the newly-created node.

The next section of the program contains the definition of each of these closures:

```
(pressed (lambda (self event)
           (seq
```

```
            (setAttr self
                     "pointF"
                     (object ("java.awt.geom.Point2D$Double"
                              (invoke event "getX")
                              (invoke event "getY"))))
            (setAttr self "draggedF" (boolean true)))))

(released (lambda (self event)
            (setAttr self "draggedF" (boolean false))))

(dragged
 (lambda (self event)
   (let ((dragon (getAttr self "draggedF")))
     (if (and (!= dragon (object null)) dragon)
         (let ((pnt (getAttr self "pointF"))
               (eX (invoke event "getX"))
               (eY (invoke event "getY"))
               (a (let ((temp (object ("java.awt.geom.AffineTransform"))))
                    (seq (invoke temp
                                 "translate"
                                 (- eX (invoke pnt "getX"))
                                 (- eY (invoke pnt "getY")))
                         temp))))
           (seq (invoke self "transform" a)
                (invoke pnt "setLocation" eX eY)
                (invoke view "repaint")))))))

(clicked (lambda (self event)
           (if (invoke event "isShiftDown")
               (let ((chooser (object
                               ("javax.swing.JColorChooser")))
                     (color (invoke chooser
                                    "showDialog"
                                    frame
                                    "Color Chooser"
                                    (invoke self "getFill"))))
                 (seq (if (!= color (object null))
                          (invoke self "setFill" color))
                      (invoke view "repaint"))))))
```

When an event handler closure is invoked it receives two arguments: the object receiving the event, and the event object itself. The closure uses information stored in the event object to manipulate the receiving object.

Each of the above closures manipulate the node object in ways that correspond to the type of event the node object received. For example, the `clicked` closure checks whether the shift key is depressed; if the key is depressed, the closure displays a dialogue box, and changes the colour of the node to the selected colour.

Next, a `java.awt.geom.AffineTransform` object is defined, which will later be used to provide translation of the node object in response to a mouse drag event.

```
(trans (let ((temp (object ("java.awt.geom.AffineTransform"))))
         (seq (invoke temp "translate" xPos yPos)
              temp))))
```

The event handling closures that have been created and bound to variables must now be registered as event handlers for the node object. The following code sequence invokes the `setMouseAction` method of the node object to register closures for each of the different types of mouse event to which the node responds.

```
(seq
 (invoke node
         "setMouseAction"
         java.awt.event.MouseEvent.MOUSE_PRESSED
         pressed)
 (invoke node
         "setMouseAction"
         java.awt.event.MouseEvent.MOUSE_RELEASED
         released)
 (invoke node
         "setMouseAction"
         java.awt.event.MouseEvent.MOUSE_CLICKED
         clicked)
 (invoke node
         "setMouseAction"
         java.awt.event.MouseEvent.MOUSE_DRAGGED
         dragged)
 (invoke view "add" node trans)))))
```

The final `invoke` expression in the preceding code sequence simply adds the translated node to the view object.

The program now defines a closure that creates a new node in response to a mouse click event.

```
(clickedV (lambda (self event)
            (if (not (invoke event "isShiftDown"))
                (seq
                 (apply mkNode
                        (- (invoke event "getX")
                           (/ w (int 2)))
```

```
                    (- (invoke event "getY")
                       (/ h (int 2))))
           (invoke view "repaint"))))))
```

The `clickedV` closure receives the window object and event object as arguments, and uses the event object to determine the position at which to create the new node. `clickedV` creates a new node object by calling the `mkNode` closure defined previously.

Since a mouse click event creates a new node, it must be received by the window object, and not a node object. Consequently, the closure that handles mouse click events is not created within the body of `mkNode`, unlike the previously defined event handler closures.

The remaining section of code simply registers the node-creation event handler, `clickedV`, with the window object, and displays an initial node.

```
(seq
 (invoke view
         "setMouseAction"
         java.awt.event.MouseEvent.MOUSE_CLICKED
         clickedV)
 (apply mkNode (* w (int 3)) (* h (int 3)))
 (invoke view "repaint")))
```

After the program has been evaluated the interpreter exits, and Java's AWT thread continues running to listen for window events. Closures that are invoked in response to events are interpreted in this second thread.

The code listing for the `clicker` program demonstrates the power and simplicity of the interface provided the `JLambda` language to Java's language facilities. Features such as closures and lexical scoping make construction of event-driven graphical programs, such as `clicker`, particularly straight-forward.

5 Related Work & Related Issues

In this section we describe briefly several projects related to `JLambda`. Additionally, we present an overview of the implementation of the `JLambda` interpreter.

5.1 Projects Related to `JLambda`

There exist several Scheme interpreters that use Java as their implementation language. Four well-known examples are SISC [14], Kawa [15], and JScheme [16], and Skij [17]. Although all of these projects aim to produce a Java-based implementation of the Scheme language, they each have unique characteristics. We present here a brief description of each project and its interface to the Java language. This is not an exhaustive list by any means, there are close to two hundred languages (200) cited in [18] that use Java or the Java Virtual Machine as a basis, roughly twenty (20) of these are classed as Scheme or Lisp like.

5.1.1 SISC

SISC is a Scheme interpreter implemented in Java whose primary goal is rapid execution of the complete Revised^5 Report on the Algorithmic Language Scheme [10] definition. In particular, SISC supports proper tail recursion and unrestricted first-class continuations. SISC's foremost concern is performance, and it outperforms all existing Java-based Scheme interpreters, often by more than an order of magnitude.

The interface to the Java language provided by SISC is the *S2J* module. Importing this module allows one to instantiate Java classes, call methods on Java objects, and access/modify fields of Java objects.

Java classes are types in SISC's extensible type system; the procedure

```
(java-class symbol)
```

returns the Java class of name `symbol`. Java classes can be instantiated with a call to the `java-new` procedure:

```
(java-new class args ...)
```

which selects a constructor of `class` based on the types of `args` and calls it, returning the newly created object.

Java fields are made accessible to Scheme code as procedures that can get/set any field of a given name on any Java object. Static Java fields can be accessed/modified by passing an instance of the appropriate class as the first argument to the procedures. The procedures

```
(generic-java-field-accessor symbol)
(generic-java-field-modifier symbol)
```

return a procedure that when invoked with a Java object as the first argument, retrieves or sets the value of the Java field named `symbol` in the Java object.

Methods are made accessible to Scheme code as procedures that can invoke any method of a given name on any Java object. For example, the procedure

```
(generic-java-method symbol)
```

returns a procedure that when invoked with a Java object as the first argument and Java values as the remaining arguments, invokes the best matching method named `symbol` on the Java object, and returns the result. Static Java methods can be invoked by passing an instance of the appropriate class as the first argument to the procedure.

5.1.2 Kawa

The goal of Kawa is a Scheme environment implemented in Java that compiles Scheme code into the byte-code instructions of the Java Virtual Machine. All of the required and optional features of the R5RS Scheme standard are supported, with the exception of proper tail recursion and unrestricted continuations. Kawa also provides mechanisms for the definition, creation, and access of Java objects.

In Kawa, objects are created with the procedure

```
(make type arg ...)
```

which constructs a new object instance of the specified `type`, which should be of the form `<java.lang.Class>`.

For accessing the fields of Java objects and static fields, Kawa provides two procedures, `field` and `static-field`, which return the value of the field. Note that the field must be declared public.

```
(field object fieldname)
(static-field class fieldname)
```

Fields may be set by using the `field` and `static-field` procedures as the first operand to `set!`. Here is an example:

```
(set! (field a 'car) (field b 'car))
```

Java instance methods and static methods may be invoked with the `invoke` and `invoke-static` procedures:

```
(invoke object name arg ...)
(invoke-static class name arg ...)
```

5.1.3 JScheme

The JScheme project is a dialect of Scheme whose chief feature is a simple and comprehensive interface to Java. JScheme supports all features of the R4RS standard except unrestricted first-class continuations and mutable strings. The interface to the Java language provided by JScheme is called the *Javadot notation*, which enables access by name to all methods, constructors, and fields of any Java class. Table 1 provides some examples of this notation.

Syntax	Type of Member	Example
"." at end	constructor	`(Font. NAME STYLE SIZE)`
"." at beginning	instance method	`(.setFont COMP FONT)`
"." at beginning, $ at end	instance field	`(.first$ '(1 2))`
".class" suffix	Java class	`Font.class`
"." only in the middle	static method	`(Math.round 123.458)`
"$" at end	static field	`Font.BOLD$`

Table 1: Examples of JScheme's Javadot notation

5.1.4 Skij

Skij, developed at IBM Watson Labs by Michael Travers, is a Scheme interpreter implemented in Java designed for exploratory programming in the Java environment. The project has now been retired by IBM, and consequently is no longer available. We mention it here because its purpose is similar to that of `JLambda`, and several of its ideas are closely related to ideas present in `JLambda`.

In Skij, Java objects are created and manipulated using the primitive procedures `new`, `invoke`, `peek`, `poke`, `invoke-static`, `peek-static`, and `poke-static`. Class arguments must be either the fully qualified class name as a string or symbol, or a `java.lang.Class` object. Method and field names should be either strings or symbols.

An object is created with the `new` procedure:

```
(new class args ...)
```

Instance fields and static fields are accessed with the `peek` and `peek-static` procedures:

```
(peek object field-name)
(peek-static class field-name)
```

Instance fields and static fields are modified with the `poke` and `poke-static` procedures:

```
(poke object field-name new-value)
(poke-static class field-name new-value)
```

Instance methods and static methods may be invoked with the `invoke` and `invoke-static` procedures:

```
(invoke object method-name args ...)
(invoke-static class method-name args ...)
```

5.2 Overview of the `JLambda` Interpreter

We provide here a brief overview of the the implementation of the `JLambda` interpreter; further details of the interpreter's design and implementation can be found in David Porter's Honours thesis [19].

The interpreter for the `JLambda` language is implemented in Java, and consists of three components: a parser, a syntax analysis phase, and an evaluation phase. Choosing Java as the interpreter's implementation language leads to certain complications in the interpreter's design. In general, the simplest way to implement an interpreter of a Scheme-like language is by using a simple recursive evaluation model in which an expression is evaluated by recursively evaluating each subexpression. If such an interpreter is implemented in Java it will exhibit recursive execution behaviour, since it inherits the control structure of the underlying Java system. However, the lack of proper tail recursion in Java means the interpreter overflows the JVM stack when attempting to evaluate arbitrarily long recursions, such as the computation of long lists.

The goal then was to design the `JLambda` interpreter so that it uses an iterative execution process; this was achieved by implementing the interpreter as a register-machine interpreter. In this design, the procedure-calling and argument-passing mechanisms used in the evaluation process are implemented in terms of operations on registers. We thus obtained an explicit-control interpreter that exhibits iterative execution behaviour.

Converting the interpreter design from a simple recursive evaluation model to a register machine interpreter involved two steps. First, we ensured all recursive calls were tail calls, by transforming the interpreter into continuation passing style [20]. If the interpreter implementation language was properly tail recursive, this transformation would have been sufficient to achieve an interpreter with iterative execution behaviour, since properly tail recursive languages guarantee that tail recursion is equivalent to iteration. However, since we chose Java as the implementation language—which is not properly tail recursive—a further step was required, in which we manually transformed tail recursion into iteration.

The second step consisted of the transformation of the interpreter from a continuation passing style into a register-based imperative style. This transformation is based on the following observation: if a set of methods call each other only by tail calls, we can first rewrite the calls to use variable assignment instead of argument-passing, and we can then replace method calls with jumps. The register machine transformation consists of systematically performing such rewrites. The result of performing the continuation passing transformation and the register machine transformation was an interpreter with iterative execution behaviour. This interpreter can therefore properly evaluate any `JLambda` expression. The transformation from recursive to iterative interpreter is a standard technique in the programming language community. An example worked out in detail can be found in Felleisen's thesis [21].

The syntax analysis phase serves to improve the interpreter's execution speed. In this phase all lexical variables in a `JLambda` program are replaced by their corresponding lexical addresses in the program structure. Such a program representation enables the evaluator to retrieve variable values directly from known addresses in the run-time environment. Implementing variable lookup operations in this way is a considerable improvement over lookup operations based on searching the environment. Consequently, the addition to the interpreter of a syntax analysis phase yields a significant increase in its execution speed.

6 Method and Constructor Resolution

The Java Reflection API is both quirky, and low level. It provides objects that represent meta-level concepts such as classes, interfaces, fields, methods, constructors, and class member modifiers. It also allows for: the ability to access or update the value of an objects field; to invoke an object's method on a given set of values, or construct a new object via calling a constructor on a given set of values. What it *does not provide* is any means of resolving what method or constructor one should use, given a certain sequence of arguments. Consequently, it is left to the `JLambda` interpreter to decide on how this should be done. In Java, method and constructor resolution uses both runtime, and static compile time information. The runtime type of the target object is used, together with the compile time types of the arguments. In an interpreted language we have no static type information to rely on, and consequently we must attempt to do the best we can with the possibly *incomplete* information (due to the presence of the `null` object reference, amongst other things) we have at hand. Namely, the runtime types of the arguments.

Compared to other implementations of interpreted languages that use Java's Reflection API, see section 5, we have adopted a rather conservative approach. This is because our aim is to provide a *simple faithful and precise* interface to Java as is possible in an untyped interpreted language. A rather bolder scheme is outlined by Michael Travers in [22, 23] and used in his language Skij [17], as well as Peter Norvig's JScheme [16].

6.1 Constructor Resolution

Evaluation of a constructor call

```
(object (<exp> <exp_1> ... <exp_N>))
```

proceeds as follows. First the arguments are evaluated from left to right, the first argument <exp> should evaluate to string representing the full name of a Java class. If the so named Java class is not `public` an exception is thrown. It is, of course, the runtime types of the values of the arguments <exp_1> ... <exp_N> that are used to determine the appropriate constructor. If an argument's value is `null` it's type is treated like a wild card object reference.

N.B. Only constructors that are declared `public` qualify in this search.

The search proceeds as follows. First the interpreter looks for a constructor that exactly matches the argument types (using the

```
public Constructor getConstructor(Class[] parameterTypes)
```

method of the java.lang.Class class). Otherwise we look among the public constructors for the best match, the candidates are chosen from those returned by the

```
public Constructor[] getConstructors()
```

method of the java.lang.Class class, *not*, for example, the

```
public Constructor[] getDeclaredConstructors()
```

method of the java.lang.Class class). The notion of best is taken to mean the *least* when taking widening, the interface hierarchy, and the class hierarchy into consideration. There may be many such choices, and presently we simply choose one, making no effort to resolve ambiguities. If no constructor is found an exception is thrown.

6.2 Method Resolution

Evaluation of a static or non static method call following form:

```
(invoke <target> <method> <exp_1> ... <exp_N>)
```

proceeds as follows. The arguments are evaluated from left to right, and then the interpreter attempts to find a method, whose name is the value of <method> (which should be a String), with the appropriate arguments. It is the method of the class that the runtime value of <target> belongs to, that is subsequently found and invoked.

If no matching method is found an exception is thrown. The search is almost identical to the one undertaken in the object construction case. It is, of course, the runtime types of the values of the arguments <exp_1> ... <exp_N> that are used to determine the appropriate method. As in the constructor case, if an argument's value is null it's type is treated like a wild card object reference.

N.B. Only methods that are declared public qualify in this search.

First the interpreter looks for a method that exactly matches the argument types (using the

```
public Method getMethod(String name, Class[] parameterTypes)
```

method of the java.lang.Class class). Otherwise we look among the public methods for the best match, the candidates are chosen from those returned by the

```
public Method[] getMethods()
```

method of the java.lang.Class class, *not*, for example, the

```
public Method[] getDeclaredMethods()
```

method of the `java.lang.Class` class). The notion of best is elaborated slightly from the constructor case to take into account the declared return types of the method. This is only used when the parameter types match exactly, and in this case the more specific return type is preferred. (See sections 8.2 and 8.4 of [9], and the API for the

```
public Method getMethod(String name, Class[] parameterTypes)
```

method of the `java.lang.Class` class). If no method is found an exception is thrown.

The reader should be warned that there is a long standing unresolved bug in the Java Reflection API [24] that has to do with access restrictions on inner class objects. Some methods, that should be accessible, are not. For example instances of `java.util.Iterator` that are returned by `java.util.Collection` instances are often unusable. We attempt to circumvent this problem by suppressing the Java language access checking, using the `setAccessible` method of the `java.lang.reflect.AccessibleObject` class, when situations like this arise.

6.3 Variations on this Theme

The main differences between the resolution process we have described here, and the one suggested by Travers, apart from memoisation, is that we attempt to obey Java's access restrictions, and restrict our attention to `public` methods and constructors. Travers on the other hand considers all possible methods, and uses the Java 1.2's class `java.lang.reflect.AccessibleObject` to suppress access checking. We, on the other hand, only resort to this to circumvent Java's Reflection API idiosyncrasies.

7 Debugging `JLambda`

There are at least three common ways of using the `JLambda` interpreter: using the read-eval-print loop, calling the interpreter directly from Java, and sending IOP's graphics2D actor a message. In the section we look at the first two.

7.1 The `JLambda` Read-Evaluate-Print Loop

The `JLambda` interpreter, encapsualted in the class `g2d.jlambda.ReadEvalPrintLoop`, is designed like the usual Lisp or Scheme style read, eval, then print loop. You can type any valid jlambda expression to be evaluated, and it will print the result of evaluation. It also understands a modicum of other queries:

```
q to quit  (or quit)
? to see these instructions
```

In `2544` we added the ability to pretty print a particular definition, as well as list in alphabetical order, the names of the definitions. One can also toggle the `FULL_DFNS` switch in `ReadEvalPrintLoop` to pretty print all definitions. The printing of the De Bruijn-ified bodies was eliminated.

```
  d to see the current definitions
  s <name> to see the definition of <name>
  u to see the current uids
  v to toggle the degree of verbosity in error reports
```

One can also launch the interpreter with an optional filename. This file should contain a single `JLambda` form to be evaluated, prior to entering the read, eval, print loop. The r-e-p loop, while portable, is distinctly primitive in flavour. It is very common amongst jlambda sufferers to invoke it in conjuction with the tecla utility `enhance`, see [25].

The abililty to tweak the verbosity of error reporting comes in handy when things go awry, and is provided by the `g2d.jlambda.Debugger` class. For the purposes of illustration there is a class in the `util` package `g2d.util.Bugs`:

```
package g2d.util;

public class Bugs {
    public static final int ZERO = 0;
    public static void loop(){
        loop();
    }
    public static int divideByZero(int x){
        return x / ZERO;
    }
}
```

This class is designed to have methods that throw unchecked exceptions, which from the point of view of the reflective interpreter simply result in failure:

```
> (sinvoke "g2d.util.Bugs" "loop")
Uncaught exception:
sinvoke: method invocation failed
        Method = public static void g2d.util.Bugs.loop()
        Target = g2d.util.Bugs
JLambda backtrace:
        computing result of sinvoke form @ line number 1

>
```

As one can imagine this is not very helpful news when debugging, so to gain more information in such a situation one can look at the internal Java exceptions that were generated. One can either do this with the built in read-eval-print loop `v` command, or programmatically by:

```
> (sinvoke "g2d.jlambda.Debugger" "toggleVerbosity")
true
>
```

To prevent seeing too much detail, we will also limit the printing of stack traces of the reported Java exceptions to a depth of 3 (the default is 15)

```
> (sinvoke "g2d.jlambda.Debugger" "setStackDepth" (int 3))
```

and then, by re-executing the problem form, we see the real cause of the problem:

```
> (sinvoke "g2d.util.Bugs" "loop")
Uncaught exception:
sinvoke: method invocation failed
       Method = public static void g2d.util.Bugs.loop()
       Target = g2d.util.Bugs

Cause[0]: (class g2d.jlambda.EvaluateError)

       stack[0] = g2d.jlambda.InvokeCont.invokeMethod(InvokeCont.java:120)
       stack[1] = g2d.jlambda.InvokeCont.computeResult(InvokeCont.java:78)
       stack[2] = g2d.jlambda.EvalArgsCont.handleReturn(EvalArgsCont.java:42)
       stack[3] = g2d.jlambda.Continuation.ret(Continuation.java:66)
.... 6 more (see g2d.jlambda.Debugger to increase depth)

Cause[1]: (class java.lang.reflect.InvocationTargetException)

       stack[0] = sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
       stack[1] = sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
       stack[2] = sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
       stack[3] = java.lang.reflect.Method.invoke(Method.java:585)
.... 10 more (see g2d.jlambda.Debugger to increase depth)

Cause[2]: (class java.lang.StackOverflowError)

       stack[0] = g2d.util.Bugs.loop(Bugs.java:7)
       stack[1] = g2d.util.Bugs.loop(Bugs.java:7)
       stack[2] = g2d.util.Bugs.loop(Bugs.java:7)
       stack[3] = g2d.util.Bugs.loop(Bugs.java:7)
.... 1021 more (see g2d.jlambda.Debugger to increase depth)
JLambda backtrace:
       computing result of sinvoke form @ line number 1

>
```

The underlying cause of the failure is stack overflow caused by the unguarded recursive call to `loop`.

7.2 Java Hooks into `JLambda`

The two most common entry points into the interpreter are provided the the classes: `g2d.jlambda.Evaluate` and `g2d.jlambda.Closure`.

The most common and user friendly entry point to the interpreter is the static method:

```
static Object evaluate(String desc)
```

which parses the given string as a `JLambda` expression, evaluates the result, and returns the result. The second most common entry point deals with the situation when one has a closure `closure` and one wants to apply it to an object `obj`, and of course get the result. This is done with a helper method of the `g2d.jlambda.Closure` class. Namely:

```
closure.applyClosure(obj);
```

There are a whole slew of `applyClosure` methods, varying on the arity of the closure, currently the upper bound is 4. Closures with more arguments than 4 tend to be curried anyway.

8 Version History of `JLambda`

- Version 1407: Added the boolean expressions `isnull` and `isobject`
- Version 1412: Antrl4 now used to parse `JLambda`.
- Version 1424: Swingless antl4 version for android built.
- Version 1534: Added implicit seqs into the syntax of `let`s, `lambda`s, `define`s, and `try`s.
- Version 2022: Added the class `g2d.jlambda.Result` as a way of throwing an aribtrary object.
- Version 2178: Added the adhoc `.class` clause for static field references.
- Version 2180: Added the ability for `for` over a integer range.
- Version 2184: Added implicit seqs into the syntax of `for`s, and `catch`es.
- Version 2.0.0 (git version)

– Added the `reduce` and `nreduce` forms.

– Made `java.lang.String` iterable in the `for`, `reduce` and `nreduce` forms.

– Added `tuples`.

– Added readable array types.

References

[1] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and C. Talcott. Maude 2.0 Manual, 2003. `http://maude.csl.sri.com/maude2-manual`.

[2] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A Tutorial Introduction to PVS. Technical report, SRI International, 1995. Presented at WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida.

[3] Saddek Bensalem, Vijay Ganesh, Yassine Lakhnech, César Mu noz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saïdi, N. Shankar, Eli Singerman, and Ashish Tiwari. An overview of SAL. In C. Michael Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pages 187–196, Hampton, VA, jun 2000. NASA Langley Research Center.

[4] I. A. Mason and C. L. Talcott. IOP: The InterOperability Platform & IMaude: An Interactive Extension of Maude. In *International Workshop on Rewriting Logic and its Applications (WRLA 2004)*, Electronic Notes in Theoretical Computer Science. Elsevier Science, 2004.

[5] Ben Funnell. The `Glyphics` Hierarchy., 2004. `http://mcs.une.edu.au/~iop/Data/Papers/`.

[6] Joel Bartlett. Ezd – easy-to-use structured graphics for Java. `http://research.compaq.com/wrl/projects/Ezd/home.html`.

[7] `http://java.sun.com/products/java-media/2D/forDevelopers/java2dfaq.html`. The Java 2D FAQ.

[8] `http://mcs.une.edu.au/~iop`. The IOP Homepage.

[9] James Gosling, Bill Joy, and Guy Steele. *Java Language Specification: The Java Series*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[10] Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.

[11] `http://mcs.une.edu.au/~iop/GraphicsActor2D/doc/`. The Graphics2D Actor API.

[12] Alan Bawden and Jonathan Rees. Syntactic Closures. In *Proceedings of the 1988 ACM Symposium of LISP and Functional Programming*, pages 86–95. ACM Press, 1988.

[13] `http://mcs.une.edu.au/~iop/Data/JLambda/Misc/clicker.lsp`. The Code Listing of the Clicker Example.

[14] Scott G. Miller. SISC: A Complete Scheme Interpreter in Java. Technical report, Indiana University, January 2002.

[15] Per Bothner. Kawa – Compiling Dynamic Languages to the Java VM. In *Proceedings of the Usenix Annual Technical Conference*, June 1998.

[16] K. Anderson, T. Hickey, and P. Norvig. SILK: A Playful Blend of Scheme and Java. In *Proceedings of the Workshop on Scheme and Functional Programming*, pages 13–22, September 2000.

[17] `http://xenia.media.mit.edu/~mt/skij/index.html`. Skij Homepage, 2004.

[18] `http://www.robert-tolksdorf.de/vmlanguages.html`. Programming Languages for the Java Virtual Machine.

[19] David Porter. An Interpreter for `JLambda`., 2004. `http://mcs.une.edu.au/~iop/Data/Papers/`.

[20] G. Plotkin. Call by Name, Call by Value, and the Lambda Calculus. *Theoretical Computer Science*, 1, 1974.

[21] M. Felleisen. *The Calculi of Lambda-v-cs Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Indiana University, 1987.

[22] Michael Travers. What is interactive scripting? *Dr Dobb's Journal*, 25:103–110, 2000.

[23] Michael Travers. Scripting and dynamic interaction in Java. Online at `http://xenia.media.mit.edu/~mt/skij/dynjava/dynjava.html`.

[24] `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4071957`. The Java Reflection API Bug.

[25] Martin Shepherd. The Tecla command-line editing library. `http://www.astro.caltech.edu/~mcs/tecla/`.