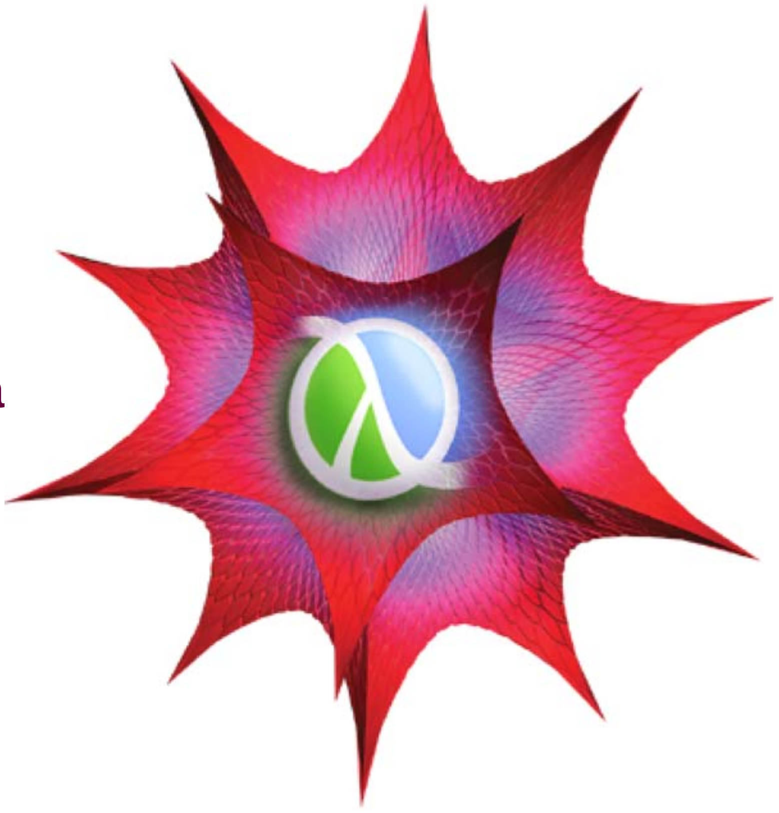
# Clojuratica



A Tutorial

**Expand each section of the tutorial by clicking the arrow to the left of the section's title.**

# First Steps

### Introduction

Clojuratica brings together two of today's most exciting tools for high-performance, parallel computation.

Clojure is a new, dynamic, functional programming language designed from the ground up for concurrency and parallelism. Wolfram *Mathematica* is arguably the world's most powerful tool for numerical computation, symbolic mathematics, optimization, and visualization and is built on top of its own splendid functional programming language.

By linking the two:

- Clojuratica creates a bidirectional interface between Clojure and *Mathematica*, enabling the **seamless translation of native data structures** between Clojure and *Mathematica*. This includes high-precision numbers, matrices, N-dimensional arrays, and evaluated or unevaluated *Mathematica* expressions and formulae.
- Clojuratica makes it easy to **evaluate Mathematica expressions from Clojure**. Now Clojure programs can take advantage of *Mathematica*'s enormous range of numerical and symbolic algorithms and fast matrix algebra routines.
- Clojuratica provides **an elegant syntax for accessing Clojure from Mathematica**, making it possible to call the methods of Clojure/Java classes without worrying about data-type conversion. Now you can run Clojure programs and instantiate Clojure classes seamlessly from *Mathematica*. All of Clojure's concurrency features are at your fingertips.
- Clojuratica lets you **call, pass, and store *Mathematica* functions just as though they were first-class functions in Clojure**. This is high-level functional programming at its finest. You can write a function in whichever language is more suited to the task and never think again about which platform is evaluating calls to that function.
- Clojuratica facilitates **the "Clojurization" of Mathematica's existing parallel-computing capabilities**. Clojuratica includes a high-performance, non-blocking concurrency queue that sits between Clojure and *Mathematica*. Now it is easy to run a simulation in Clojure with 10,000 independent threads asynchronously evaluating processor-intensive expressions in *Mathematica*. The computations will be farmed out adaptively and transparently to however many *Mathematica* kernels are available on any number of processor cores, either locally or across a cluster, grid, or network.

Clojuratica is open-source and targeted at applications in scientific computing, computational economics, finance, and other fields that rely on the combination of parallelized simulation and high-performance number-crunching. Clojuratica gives the programmer access to Clojure's most cutting-edge features — easy concurrency and multithreading, immutable persistent data structures, and software transactional memory (more on Clojure) — alongside *Mathematica*'s world-class algorithms for numerics, symbolic mathematics, optimization, statistics, visualization, and image-processing (more on *Mathematica*).

The canonical pronunciation of *Clojuratica* starts with *Clojure* and rhymes with *erotica*.

## Installation

### The Clojure Library

Clojuratica has two components: the Clojure library and the *Mathematica* package. The Clojure library will function whether or not the *Mathematica* package is installed. The *Mathematica* package, however, requires that the Clojure library be installed.

The Clojure library provides access to *Mathematica* from Clojure and is contained in the java archive (JAR) file clojuratica.jar.

The Clojure library's JAR file must be installed in the system's Java classpath for either the Clojure library or the *Mathematica* package to operate.

To install the JAR file in your system's classpath, download the file and place it in a directory contained in your system's Java classpath. Alternatively, place it in an arbitrary location and edit your system's classpath appropriately. Typically one edits the classpath by changing a system environmental variable. The internet offers countless resources on how to access and edit the Java classpath.

The *Mathematica* J/Link library, JLink.jar, must also be in your system's classpath for either the Clojure library or the *Mathematica* package to work. JLink.jar is typically located in the AddOns/JLink subdirectory of the Mathematica installation. Some Mathematica versions will by default not install J/Link. If this is the case with yours, you can download J/Link for free from the Mathematica J/Link web site.

Before doing anything else, make sure clojuratica.jar and JLink.jar are in the Java classpath.

Note: Mathematica has its own classloader. It will not be able to locate JAR files whose entry in the classpath consists of a directory and a wildcard asterisk (C:/Users/username/jar/*), even though Java allows this. You must enter the full path to the JAR.

#### The *Mathematica* Package

The *Mathematica* package provides access to Clojure from *Mathematica* and is contained in the *Mathematica* package file Clojuratica.m.

The *Mathematica* package requires that the Clojure library be installed. See the previous section for details.

To install the *Mathematica* package, download the file Clojuratica.m and place it in a directory in the *Mathematica* path. Alternatively, place it in an arbitrary location and edit the *Mathematica* path appropriately. To view the *Mathematica* path, evaluate the $Path variable in a *Mathematica* notebook. *Mathematica*'s documentation contains resources about editing the path.

# The Clojure Library

## Getting Started

#### Context

Clojuratica's Clojure library offers easy access to *Mathematica* from Clojure and facilitates a Clojure-centric programming paradigm. In this paradigm, one uses Clojure to write the program's control-flow, making calls to *Mathematica* when necessary.

In the Clojure-centric paradigm, the Clojure code can take the form of Java classes compiled using Clojure's gen-class functionality, uncompiled .clj files, or a mixture of the two.

A *Mathematica*-centric paradigm is also possible using Clojuratica's *Mathematica* package. This paradigm is discussed in the second half of the tutorial.

#### Basic Setup

Getting underway, let's use the Clojuratica package.

```
=> (use 'clojuratica.clojuratica)
nil
```

Clojuratica needs to have access to a *Mathematica* kernel running locally or remotely. The simplest way to provide this access is to launch a local kernel from within Java. The procedure will vary slightly by platform and is dictated by the *Mathematica J/Link* package. On Windows, the following commands are appropriate. See the *Mathematica J/Link* user guide for more details. Note that these commands have nothing to do with Clojuratica per se. They are general to any Java program that accesses *Mathematica*.

```
=> (import '[com.wolfram.jlink MathLinkFactory])
nil

=> (def kernel-link (MathLinkFactory/createKernelLink
                      (str "-linkmode launch -linkname "
                           "'c:/program files/wolfram research/"
                           "mathematica/7.0/mathkernel.exe'")))
#'user/kernel-link

=> (.discardAnswer kernel-link)   ;discard the prompt given by the kernel
nil
```

Assuming the above code did not throw a MathLinkException, we now have an active kernel. Note that only one or two master kernels can be active at once under most personal *Mathematica* licenses. If you receive an error, it may be because you attempted to launch a kernel when another kernel or two were already running. (In addition to the master kernel, up to four parallel kernels can usually be active; parallel computation is discussed later.)

## Calling Mathematica

### The Evaluator

The basic Clojuratica function is the **evaluator**. Getting an evaluator is easy.

```
=> (def evaluate (get-evaluator kernel-link))   ;get an evalautor that knows
where the Mathematica kernel is
#'user/evaluate
```

The evaluator is all we need to do some math.

```
=> (evaluate [] "1 + 1")   ;evaluate the Mathematica expression 1+1
(#<Expr Integer>)
```

The evaluator's syntax is similar to that of Clojure's let. The first argument is a vector of assignments, empty here. The remaining arguments are *Mathematica* expressions to be evaluated.

The output of the evaluator is a Java object of class CExpr. CExpr stands for Clojuratica expression. It is a wrapper class for *Mathematica* expressions. CExpr objects can hold any *Mathematica* expression. An expression can be as simple as a single integer or as complex as a thousand operations on a hundred-dimensional array.

Remember that *Mathematica* expressions, and therefore CExprs, are homoiconic: they can house "pure data" (integers, lists, vectors, matrices, etc.), symbols for symbolic math (x, y, etc.), and unevaluated mathematical expressions containing operations and functions ("1+1", "x+y", "FactorInteger[15]", etc.).

The following demonstrates that the output of the evaluator is a CExpr:

```
=> (class (evaluate [] "1 + 1"))
clojuratica.CExpr
```

#### The Parser

We can easily convert the CExprs we get as evaluator output into Clojure data types and data structures. To do this we'll need a Clojuratica **parser**.

```
=> (def parse (get-parser kernel-link))           ;get a Clojuratica parser
that knows where the Mathematica kernel is
#'user/parse

=> (parse (evaluate [] "1 + 1"))                  ;parse the result of
evaluating 1+1
2

=> (parse (evaluate [] "3x + 7 + 2x - 5x"))       ;remember, this is
Mathematica: symbolic math works too
7

=> (parse (evaluate [] "3x + 7 + 2x - 5x + z"))   ;output that cannot be
parsed fully remains in CExpr form
(#<Expr Plus> #<Expr 7> #<Expr z>)

=> (class *1)
clojuratica.CExpr
```

Take note of the structure of the second-to-last return value above. Like Clojure expressions, *Mathematica* expressions use prefix notation internally. The operator, Plus, is at the head of the expression, followed by the arguments 7 and z.

The parser can take a string as input instead of a CExpr. In this case it will parse the string as an expression without evaluating it.

```
=> (parse "2 + 5")  ;parse without evaluating
(#<Expr Plus> #<Expr 2> #<Expr 5>)
```

Note that the 2 and 5 remain separate. If evaluated they would sum to 7. You can store the unevaluated CExpr in your Clojure program and later send it to *Mathematica* for evaluation. As the following shows, the evaluator accepts CExpr objects just like it accepts strings.

```
=> (evaluate [] *1)
(#<Expr Integer>)

=> (parse *1)
7
```

The 2 and 5 have now been summed.

Parse and evaluate are often used together. It is useful to compose them.

```
=> (def math (comp parse evaluate))
#'user/math

=> (math [] "3x + 7 + 2x - (10/2)x")  ;pretty syntax
7
```

Keep in mind that the evaluator accepts multiple expressions:

```
=> (math []
     "foo = 3x + 7 + 2x - (10/2)x"
     "foobar = foo + 3"
     "foobar ^ 2")
100
```

The foregoing could also be written using *Mathematica* compound-expression syntax:

```
=> (math []
     "foo = 3x + 7 + 2x - (10/2)x;
      foobar = foo + 3;
      foobar ^ 2")
100
```

## Working with Data Structures

### Lists, Matrices, and Arrays

The parser knows how to handle *Mathematica* lists and arrays. They are converted to Clojure data structures on the fly.

```
=> (math [] "{{1, 0},
              {0, 1}} * 5")   ;parse the result of multiplying the 2x2 identity
matrix by 5
((5 0) (0 5))
```

The parser converts *Mathematica* lists to lazy seqs. Two-dimensional arrays (aka matrices) are converted to lazy seqs of lazy seqs. And so on for higher-dimensional arrays.

```
=> (class *1)
clojure.lang.LazySeq
```

Clojuratica has no problem parsing high-dimensional *Mathematica* arrays. The following *Mathematica* code creates an eight-dimensional array, two elements in each direction, populated with ones:

```
=> (math [] "ConstantArray[1, {2, 2, 2, 2, 2, 2, 2, 2}]")
((((((((1 1) (1 1)) ((1 1) (1 1))) (((1 1) (1 1)) ((1 1) (1 1 ... truncated ...
```

### Assignments: Sending Data to Mathematica

Recall that the evaluator permits assignments in its vector first argument. That means we can write:

```
=> (math ["foo" 2] "foo + 10")   ;evaluate and parse 2+10
12
```

The assigned objects can be Clojure data structures. They are automatically converted to *Mathematica* data structures where appropriate.

```
=> (math ["foobar" [[1 0]
                    [0 1]]]
     "foobar * 2")
((2 0) (0 2))
```

Clojuratica handles large numbers without loss of precision, using as little memory as possible.

```
=> (math ["bignum"
584750243857042857432059486084257428574395 8.4329874329487329487324445309854309574302750296843750M]
     "bignum + 1")
5847502438570428574320594860842574285743959.43298743294873294873244453098543095743027502968437 5M
```

Variables set in the evaluator's binding vector are lexically scoped within the evaluator call.

```
=> (math [] "bignum")
(#<Expr Symbol>)
```

*Mathematica* returned the symbol bignum rather than the number above, showing that bignum was defined only in the scope of the evaluator call that bound it. Internally, Clojuratica wraps each call to the evaluator in a *Mathematica* Module[] and executes the module. Modules implement lexical scoping.

### The Parser is Lazy

When parsing *Mathematica* lists and arrays, the seq returned by the parser is lazy. This is helpful for handling large data structures. Elements will be parsed just-in-time.

For instance, the following Mathematica expression creates a list with 100,000 elements.

```
=> (def long-list (math [] "Table[i^2, {i, 100000}]"))
#'user/long-list
```

Taking the first 10 elements is fast.

```
=> (time (take 10 long-list))
"Elapsed time: 0.670057 msecs"
(1 4 9 16 25 36 49 64 81 100)
```

Whereas running through all the elements takes time:

```
=> (time (nth long-list 99999))
"Elapsed time: 8386.220665 msecs"
10000000000
```

Since this is a lazy seq, the data-structure is cached. After the first access, access times are constant.

```
=> (time (nth long-list 99999))
"Elapsed time: 2.406241 msecs"
10000000000
```

Clojure never ceases to amaze.

### Parsing Directly to Vectors

If you want constant access times from the get-go, you can tell the parser to parse to vectors instead of lazy seqs. Generally lazy seqs will be the better choice, but vectors have their place. The parser accepts the flag :vectors. The flag can go anywhere among the arguments to the parser.

```
=> (def long-list-cexpr (evaluate [] "Table[i^2, {i, 100000}]"))   ;as before, 
generate a long list of numbers
#'user/long-list-cexpr

=> (time 
     (def long-list-lazyseq (parse long-list-cexpr)))              ;parsing to 
lazy seqs, as before, is fast
"Elapsed time: 274.136587 msecs"
#'user/long-list-lazyseq

=> (time 
     (def long-list-vector (parse :vectors long-list-cexpr)))      ;parsing to 
vectors takes longer...
"Elapsed time: 6551.951924 msecs"
#'user/long-list-vector

=> (time (dorun long-list-vector))     ;...but access times for the vector are 
constant from the start
"Elapsed time: 21.763451 msecs"
nil

=> (time (dorun long-list-lazyseq))    ;whereas, as we saw earlier, the first 
traversal of the lazy seq is slow
"Elapsed time: 8667.238434 msecs"
nil
```

High-dimensional data-structures are parsed to vectors without any fuss, just as with parsing to lazy seqs.

```
=> (parse :vectors (evaluate [] "ConstantArray[1, {2, 2, 2, 2, 2, 2, 2, 2}]"))
;create and parse-to-vectors an 8D array
[[[[[[[[1 1] [1 1]] [[1 1] [1 1]]] [[[1 1] [1 1]] [[1 1] [1 1 ... truncated ...
```

## Changing Clojuratica's Behavior

### The Flag System

In the previous section we encountered our first flag, :vectors, which told the parser to parse to vectors instead of lazy seqs. Many Clojuratica functions accept flags that change their behavior. We will encounter more as we go on.

In the last section, calling the parser with a flag required that we eschew the use of the composed function `math`, for the flag was a flag to the parser, not the evaluator. This was a pain. Instead of doing the natural thing and writing:

```
(math :vectors [] "{1, 2, 3}")
```

We were forced to break apart the two calls and write:

```
(parse :vectors (evaluate [] "{1, 2, 3}"))
```

In fact, it turns out that we *can* write the first form!

```
=> (math :vectors [] "{1, 2, 3}")
[1 2 3]
```

This works because the evaluator accepts any flag at all. Flags the evaluator does not recognize will be attached to the CExpr it returns. When this CExpr is later passed to a parser, the parser looks for any attached flags and behaves appropriately.

```
=> (def foo (evaluate :vectors [] "{1, 2, 3}"))   ;the :vectors flag attaches
to foo
#'user/foo

=> (parse foo)   ;later, the parser acts in accordance with the attached flags
[1 2 3]

=> (parse :seqs foo)   ;but you can override attached flags if you like
(1 2 3)
```

Accordingly, you can pass parser flags to the composed function `math`. Any parser flags will attach to the return value of the evaluator and then be recognized by the parser.

Note that if multiple conflicting flags are passed to a function, only the first is honored.

```
=> (parse :seqs :vectors :seqs foo :vectors)
(1 2 3)
```

### Defaults

The basic function generators (get-parser, get-evaluator, etc.) all accept flags. Any flags specified during function generation will be retained and will modify the default behavior of the generated function.

```
=> (def parse (get-parser :vectors kernel-link))
#'user/parse

=> (parse (evaluate [] "{1, 2, 3}"))
[1 2 3]

=> (def parse (get-parser :seqs kernel-link))
#'user/parse

=> (parse (evaluate [] "{1, 2, 3}"))
(1 2 3)
```

### Example: Requesting Output for All Expressions in an Evaluator Call

By passing the :all-output flag to the evaluator, you can tell it to return the result of every expression in the call, not just the last. The flag can go anywhere among the arguments.

```
=> (math [] :all-output
     "foo = 3x + 7 + 2x - (10/2)x"
     "foobar = foo + 3"
     "foobar ^ 2")
(7 10 100)
```

As explained earlier, the :all-output flag can also be passed to get-evaluator, in which case it will be retained as the default.

```
=> (def evaluate* (get-evaluator kernel-link :all-output))
#'user/evaluate*

=> (def math* (comp parse evaluate*))
#'user/math*

=> (math* []
     "foo = 3x + 7 + 2x - (10/2)x"
     "foobar = foo + 3"
     "foobar ^ 2")
(7 10 100)
```

The opposite of the :all-output flag is the :last-output flag. Calling math* with the :last-output flag would override the default :all-output flag.

## Thread-Safety and Global Variables

### Clojuratica is Thread-Safe

Rest assured that calls to the evaluator executed in different threads will not interact.

Let's define a function containing some Mathematica code that would act erratically if evaluated simultaneously in different threads (if Clojuratica were not thread-safe).

```
=> (defn count-up [] (math ["counter" 0]
                       "Do[counter++, {500000}]"
                       "counter"))
#'user/count-up
```

Executing the function in four concurrent threads, we find that the counter variable acts reliably.

```
=> (pvalues (count-up) (count-up) (count-up) (count-up))
(500000 500000 500000 500000)
```

If Clojuratica were not thread-safe, we would expect larger return values.

### The Global Setter

Variables defined in the assignment vector of the evaluator are lexically scoped within that call to the evaluator. However, any variables defined in Mathematica expressions have global scope unless you scope them yourself. This is a feature, not a bug, but be careful!

```
=> (defn count-up2 [] (math [] "Do[counter++, {500000}]"      ;this time we do
not set the counter to 0
                               "counter"))
#'user/count-up2

=> (evaluate [] "counter = 0")                                ;define counter---
in a global scope!
(#<Expr Integer>)

=> (pvalues (count-up2) (count-up2) (count-up2) (count-up2))  ;probably the
"wrong" result!
(500000 1500000 1000000 2000000)
```

Global Mathematica variables can be set explicitly using the **global setter** function. The get-global-setter function takes an evaluator as its first argument.

```
=> (def global-set (get-global-setter evaluate))
#'user/global-set
```

It is good practice to use the global setter whenever you are setting globals you care about, so as to avoid confusion like that of count-up2.

```
=> (global-set "year" 2009)
(#<Expr Integer>)

=> (math [] "year")
2009
```

Another reason to use the global setter is that it will accept arbitrary Clojure data structures. The global setter is the only way to store an arbitrary Clojure data structure as a global Mathematica variable. (Reminder: To store a Clojure data structure as a lexically scoped variable, use the assignment vector of the evaluator.)

```
=> (global-set "recentyears" [2009 2008 2007 2006])
(#<Expr List> #<Expr 2009> #<Expr 2008> #<Expr 2007> #<Expr 2006>)

=> (math [] "recentyears - 2000")
(9 8 7 6)
```

A third good reason to use the global setter is that it automatically distributes assignments to all parallel kernels whenever the evaluator with which it was created was a parallel evaluator. Which brings us to...

## Parallel Computation

### Introduction

We saw above that the Clojuratica evaluator is thread-safe. In fact, thread-safety is just the beginning.

Clojuratica can also parallelize. It can distribute *Mathemtaica* computations from multiple Clojure threads to multiple *Mathematica* kernels, which may be running on multiple cores or even multiple machines across a cluster or grid. This is possible because Clojuratica adds a thread-ready concurrency layer to *Mathematica*'s built-in parallelism capabilities.

Using Clojuratica's parallel evaluator, you can have, say, 10,000 Clojure threads all sending different expressions to *Mathematica* to evaluate. Clojuratica will adaptively farm each expression out to the next available *Mathematica* kernel and ensure that the right *Mathematica* output is returned to the right function call in the right thread.

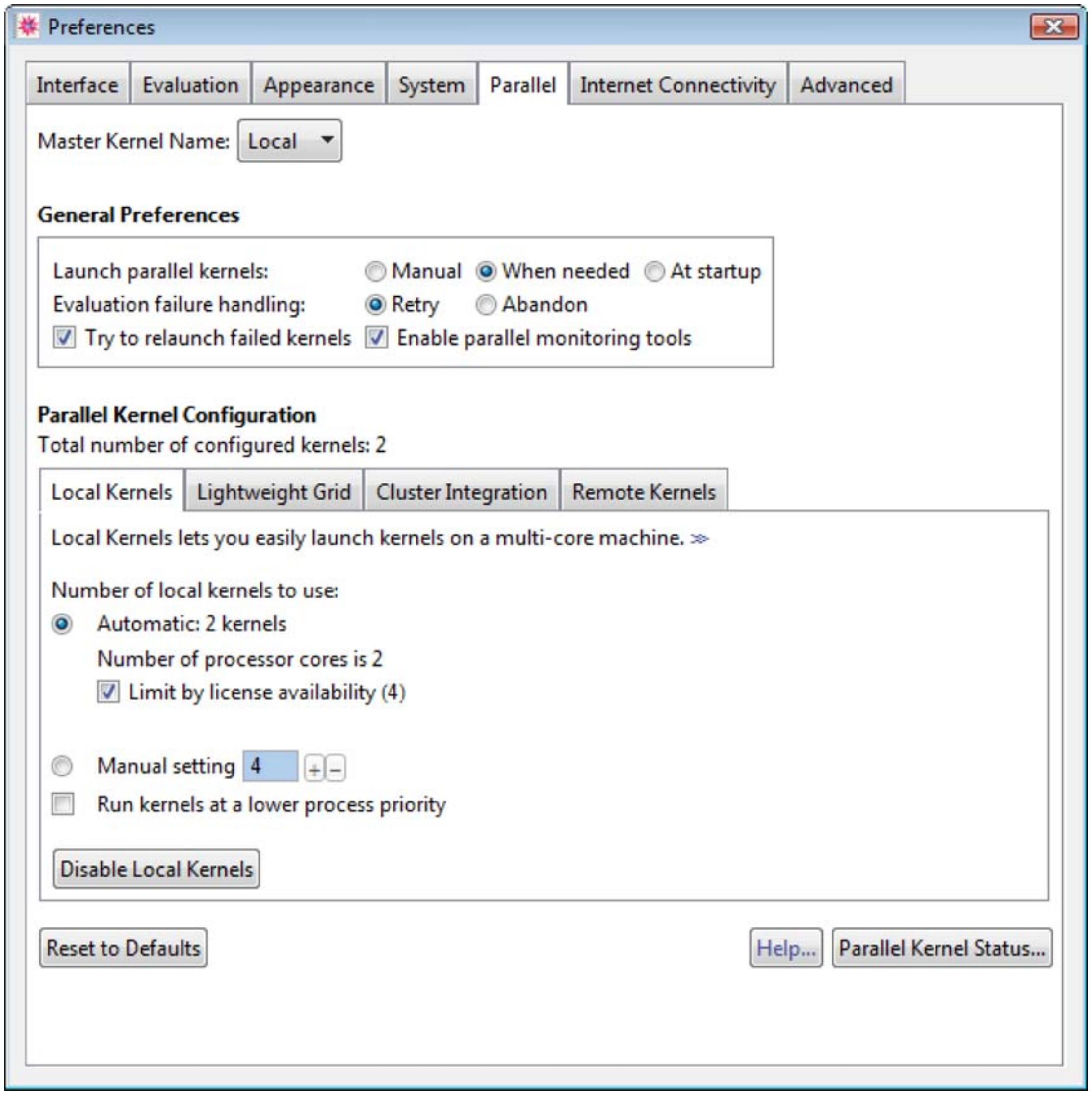
The parallel evaluator has two major selling points. First, kernels are never idle while there are expressions waiting to be evaluated. Second, the parallel evaluator's concurrency queue is robust to having fewer kernels than threads. Threads share kernels adaptively. This latter feature is important because *Mathematica*'s kernels are memory-heavy and limited in number by the availability of *Mathematica* licenses.

On a hypothetical 50-core machine we would ideally have 50 *Mathematica* kernels available. The Clojuratica concurrency queue will ensure that every expression evaluated by one of our 10,000 Clojure threads is farmed out to one of the 50 kernels. While a thread waits for its computation to finish, that thread is blocked. The queue, however, is never blocked. Other threads can send expressions to *Mathematica* at any time. A CPU-heavy computation from one thread will not block a CPU-light computation from a different thread from finishing on a different kernel.

*Mathematica* has support for an arbitrarily large number of kernels running on local cores or linked over a network. The number of kernels is limited only by the number of licenses available. This means that, in theory, you could have tens of thousands of Clojure threads evaluating thousands of expressions per minute, all being farmed out to hundreds of *Mathematica* kernels across a cluster or grid.

### Setup

Because Clojuratica's concurrency queue is built on top of *Mathematica*'s built-in parallelism capabilities, the setup of the parallel *Mathematica* kernels is done in *Mathematica* itself:



Master Kernel Name: Local

**General Preferences**

Launch parallel kernels: Manual [✓ When needed] At startup
Evaluation failure handling: [✓ Retry] Abandon
✓ Try to relaunch failed kernels ✓ Enable parallel monitoring tools

**Parallel Kernel Configuration**
Total number of configured kernels: 2

Local Kernels lets you easily launch kernels on a multi-core machine. »

Number of local kernels to use:
[✓ Automatic: 2 kernels]
Number of processor cores is 2
✓ Limit by license availability (4)

Manual setting 4
Run kernels at a lower process priority

The same configuration dialog allows the configuration of remote kernels across a cluster or the internet.

### The Parallel Evaluator

The Clojuratica evaluator we created earlier in the tutorial will not parallelize by default. To get a **parallel evaluator**, use the :parallel keyword anywhere in the call to get-evaluator:

```
=> (def pevaluate (get-evaluator kernel-link :parallel))   ;get a parallel
evaluator
#'user/pevaluate

=> (def pmath (comp parse pevaluate))   ;compose a new convenience function
#'user/pmath
```

The parallel evaluator has exactly the same syntax as the serial (regular) evaluator:

```
=> (pevaluate [] "1 + 1")
(#<Expr Integer>)

=> (pmath [] "1 + 1")
2
```

### Example 1: A Performance Test

Now we'll test whether parallel evaluation works faster than serial evaluation on my dual-core machine.

First, let's create a vector of thirty agents. These agents are going to end up storing the results of the thirty CPU-intensive computations we will farm out to thirty Clojure threads. In turn, the Clojure threads will each evaluate an expression using *Mathematica*. Clojuratica will ensure that the *Mathematica* computations are run in parallel.

```
=> (def output-vector
     (vec (for [i (range 30)] (agent nil))))  ;create a vector of thirty agents
#'user/output-vector
```

The expression we are going to evaluate thirty times over is a simple integer factorization. Thirty serial executions of this computation take about 45 seconds on my dual-core laptop.

```
=> (def expression
"FactorInteger[420394832044320948329323179131817135013171305111]")
#'user/expression

=> (math [] expression)
((7 1) (47 1) (2091308860889 1) (2947175658767 1) (207318124537824409931 1))

=> (time (doseq [x (range 30)] (math [] expression)))
"Elapsed time: 44209.43416 msecs"
nil
```

To evaluate the expression thirty times in parallel, we can simply send-off each agent with an instruction to update itself with the result of a call to the parallel evaluator. Clojure farms out the "send-off" instructions to separate threads. This means that the parallel evaluator is receiving calls from thirty different threads at once.

```
=> (doseq [output-agent output-vector]                          ;for each agent
in the output vector,
     (send-off output-agent (fn [_] (pmath [] expression))))   ;update the
agent with the result of the expression
nil
```

We can time how long it takes for all thirty expressions to complete by monitoring the output vector. When it has no more nils in it, all thirty expressions have completed. I ran the following command immediately after the doseq command above. (If you want to be really precise with the timing, you can enclose both commands in a do block.)

```
=> (time
     (while (some #(nil? @%) output-vector)    ;while any of the agents
contains a nil,
       (Thread/sleep 50)))                     ;wait for 50 more msecs
"Elapsed time: 27075.338346 msecs"
```

Since 27 seconds is 60% of 45 seconds, it's clear that parallelization is effective by almost a factor of two on my dual-core machine.

You need not use agents and send-off to parallelize. You could also use Java's Thread class directly, or use Clojure's pvalues function or similar.

Agents are, however, an elegant and simple way to make use of multiple threads in Clojure.

### Example 2: A Test of Per-Call Overhead

Keep in mind that the concurrency queue can handle evaluations from an arbitrarily large number of threads. The overhead for each evaluation is quite low. Let's evaluate 10,000 CPU-light expressions.

Create a vector of agents to store the results of 10,000 computations:

```
=> (def output-vector
     (vec (for [i (range 10000)] (agent nil))))
#'user/output-vector
```

We are going to execute Sqrt[i] 10,000 times, where i is the index of the current agent in the output vector.

```
=> (doseq [i (range (count output-vector))]            ;for each agent in the
output vector
     (send (nth output-vector i)                       ;update the agent with
the result of
           (fn [_] (pmath ["i" i] "Sqrt[i] // N"))))    ;the sqrt of the index
of that agent.
                                                       ;note that //N is
Mathematica shorthand for
                                                       ;"evaluate numerically
rather than symbolically"
nil
```

We use send instead of send-off because my machine doesn't have memory for 10,000 threads. A powerful multicore machine or Terracotta cluster might. send updates its agent using a thread from a limited thread pool. The threads are recycled, and at a certain point send will wait for a thread from the thread pool to become available rather than create a new thread. send-off always creates a new thread when there are no idle threads in the thread pool. send-off can therefore create an arbitrarily large number of threads, possibly eating up all the JVM's memory.

Wait for the results:

```
=> (time
     (while (some #(nil? @%) output-vector)    ;while any of the agents
contains a nil,
       (Thread/sleep 50)))                     ;wait for 50 more msecs
"Elapsed time: 33172.877977 msecs"
```

This result implies that the total overhead per computation is around 3.3 ms. We computed 10,000 expressions and the entire task took 33,000 ms. Check that the right thing has happened:

```
=> @(nth output-vector 2500)        ;good, 50 is the square root of 2500
50.0
```

Note that each separate call to the Clojuratica parallel evaluator will be sent to a *single* kernel. Parallelism occurs when (and only when) *multiple* threads are making calls to the parallel evaluator simultaneously. If you need to parallelize a *single* Mathematica expression, such as a call to Mathematica's Map[], you should use Mathematica's built-in parallelization functions such as Parallelize[]. In this event you need not use a parallel evaluator, although you may.

```
=> (math [] "Parallelize[Map[FactorInteger, {1, 11, 111, 1111, 11111}]]")
(((1 1)) ((11 1)) ((3 1) (37 1)) ((11 1) (101 1)) ((41 1) (271 1)))
```

### Global Variables and Parallelism

The global setter automatically distributes global variables to all parallel kernels when it was created with a parallel evaluator.

```
=> (def pglobal-set (get-global-setter pevaluate))

=> (pglobal-set "identity" [[1 0] [0 1]])
(#<Expr List> #<Expr {1,0}> #<Expr {0,1}>)

=> (pvalues (pmath [] "identity * 2") (pmath [] "identity / 2 // N"))
(((2 0) (0 2)) ((0.5 0.0) (0.0 0.5)))
```

## *Mathematica* Functions as First-Class Clojure Functions

### The Function Wrapper

The evaluator and parser jointly provide an elegant, simple interface to *Mathematica*. There are times, however, when an even tighter interface is called for.

Clojuratica provides an easy way to wrap *Mathematica* functions in a Clojure interface layer. Thereafter, they may be treated as first-class Clojure functions. All of Clojure's functional programming features instantly apply to functions written in *Mathematica*.

When a wrapped *Mathematica* function is called in Clojure, Clojuratica transparently evaluates the function call using *Mathematica*. Arguments to the wrapped function are transparently converted to *Mathematica* data types and passed to the *Mathematica* function. The return value of the *Mathematica* function is automatically converted back to a Clojure data type.

Let us illustrate wrapping a *Mathematica* function and calling it in Clojure.

First, we must get a *Mathematica* function wrapper.

```
=> (def fn-wrap (get-fn-wrapper evaluate))
#'user/fn-wrap
```

The function get-fn-wrapper returns a *Mathematica* function wrapper. get-fn-wrapper takes an evaluator as its argument. The evaluator passed to get-fn-wrapper will be used to execute calls to the wrapped *Mathematica* function.

Now we can create a wrapped *Mathematica* function. The syntax of the function wrapper is similar to that of the evaluator. We will create a function taking one argument and returing its argument incremented by one.

```
=> (def increment (fn-wrap [] "Function[{x}, x + 1]"))
#'user/increment

=> (increment 1)
2
```

It's as simple as that. Now you can treat increment as a normal Clojure function.

```
=> (map increment [3 4 5 [1 2] 6])
(4 5 6 (2 3) 7)
```

Note that the function looks and acts like a Clojure function from Clojure's perspective, but functions like a *Mathematica* function. The internal vector [1 2] is incremented because in *Mathematica* the Plus operation is automatically threaded across lists (i.e. has the Listable attribute).

Finally, note that you can specify a bare symbol as the function definition. In this case, Clojuratica will assume that the symbol is a function and call it on the passed arguments. This is useful for wrapping pre-existing *Mathematica* functions, for example.

```
=> (def factorize (fn-wrap [] "FactorInteger"))
#'user/factorize

=> (factorize 21)
((3 1) (7 1))
```

### Creating Closures

You may have noticed that the function wrapper's first argument is a vector. In the example above the vector was empty.

As with the evaluator, this vector is a vector of assignments. The function created by the function wrapper will be closed over the variables specified in the assignment vector. If you are not familiar with the notion of a closure, Wikipedia has a good introduction.

Let's close over a matrix passed as a vector of vectors.

```
=> (def identity-2-by-2 [[1 0]
                         [0 1]])
#'user/identity-2-by-2

=> (def diagonalize-2-by-2 (fn-wrap ["identity" identity-2-by-2] "Function[{x},
identity * x]"))
#'user/diagonalize-2-by-2

=> (math [] "identity = {{0, 0}, {0, 0}}")   ;this definition will not affect
the closed-over value
((0 0) (0 0))

=> (diagonalize-2-by-2 [[5 6]
                        [7 8]])
((5 0) (0 8))
```

### Many Ways to Define Functions

*Mathematica* offers a number of syntaxes for defining functions. Some are based on the anonymous function syntax ("MyIncrement = Function[{x}, x + 1]", whose shorthand is "MyIncrement = # + 1 &"). Others are based on the pattern-matching syntax ("MyIncrement[x_] := x + 1").

You can use any of these syntaxes when creating wrapped functions using the function wrapper. The following examples illustrate use of the different syntaxes.

```
=> (def increment (fn-wrap [] "Function[{x}, x + 1]"))              ;anonymous
function syntax
#'user/increment

=> (increment 1)
2
```

When using anonymous function syntax, you may, but need not, assign the anonymous function to a *Mathematica* symbol. If you do assign the function to a *Mathematica* symbol, the function name will have global scope and will be callable using the evaluator.

```
=> (def increment (fn-wrap [] "Increment1 = Function[{x}, x + 1]"))  ;anonymous
function syntax with an assignment
#'user/increment

=> (increment 1)
2

=> (math [] "Increment1[1]")
2
```

Note that the *Mathematica* symbol to which the *Mathematica* function is assigned (Increment1 in the case above) need not correspond to the name of the Clojure symbol to which the wrapped function is assigned. In this case I chose Increment1 because Increment is the name of a built-in *Mathematica* function.

When using the pattern-matching syntax, you must naturally name the function, since otherwise *Mathematica* will have no symbol to attach the pattern to. Again, the name of the function will have global scope in *Mathematica*.

```
=> (def increment-by-two (fn-wrap [] "Increment2[x_] := x + 2"))     ;pattern-
matching syntax
#'user/increment-by-two

=> (increment-by-two 1)
3

=> (math [] "Increment2[1]")
3
```

You can use the pattern-matching syntax to overload functions. Let's overload the Increment2 function such that it does something different when its argument is a string.

```
=> (def increment-by-two (fn-wrap [] "Increment2[x_String] := x <> \" plus
two\""))
#'user/increment-by-two

=> (increment-by-two 1)
3

=> (increment-by-two "1")
"1 plus two"
```

Note that we simply defined the new wrapped function right over the old one. The overloading takes place on the *Mathematica* side.

As veteran users of *Mathematica* will know, the pattern-matching syntax provides more features in complex situations than the anonymous-function syntax does (overloading, for one). In the case of creating a simple increment function, however, there is no concrete advantage to using pattern-matching syntax.

Variable arity is supported. So is recursion.

```
=> (def increment-by-two (fn-wrap [] "Increment2[x___] := Map[Increment2,
{x}]"))
#'user/increment-by-two

=> (increment-by-two "1" 6 12 "foo")
("1 plus two" 8 14 "foo plus two")

=> (increment-by-two)
nil
```

### Flags

The function wrapper automatically parses the output of the wrapped function. This behavior can be changed with the :no-parse flag.

```
=> (def increment (fn-wrap :no-parse [] "Function[{x}, x + 1]"))
#'user/increment

=> (increment 1)
(#<Expr Integer>)

=> (parse (increment 1))
2
```

As always, flags can be specified to the generator function in order to set defaults.

```
=> (def fn-wrap* (get-fn-wrapper :no-parse evaluate))
#'user/fn-wrap*

=> (def increment (fn-wrap* [] "Function[{x}, x + 1]"))
#'user/increment

=> (increment 1)
(#<Expr Integer>)
```

The opposite of :no-parse is :parse.

```
=> (def increment (fn-wrap* :parse [] "Function[{x}, x + 1]"))
#'user/increment

=> (increment 1)
2
```

The function wrapper accepts all evaluator flags (and therefore all parser flags). For example:

```
=> (def increment-list (fn-wrap :seqs [] "Function[{x}, Map[#+1&, x]]"))
;parsing to seqs
#'user/increment-list

=> (increment-list [1 2 3])   ;creates a seq
(2 3 4)

=> (def increment-list* (fn-wrap :vectors [] "Function[{x}, Map[#+1&, x]]"))
;parsing to vectors this time
#'user/increment-list*

=> (increment-list* [1 2 3])   ;creates a vector
[2 3 4]
```

### Parallel Computation Using Wrapped Functions

If you want calls to your wrapped functions to be evaluated in parallel, simply ensure that you call get-fn-wrapper with a parallel evaluator when creating your function wrapper.

```
=> (def pfn-wrap (get-fn-wrapper pevaluate))
#'user/pfn-wrap
```

Let's replicate using the function wrapper an example from the Parallel Computation section of the tutorial earler (Example 1: A Performance Test).

First, we'll create a vector of thirty agents. These agents are going to end up storing the results of the thirty CPU-intensive computations we are going to farm out to thirty Clojure threads. In turn, the Clojure threads will each evaluate a wrapped *Mathematica* function. Since our function wrapper, pfn-wrap, was created with a parallel evaluator, Clojuratica will ensure that the *Mathematica* computations are run in parallel.

```
=> (def output-vector
     (vec (for [i (range 30)] (agent nil))))  ;create a vector of thirty agents
#'user/output-vector
```

Our wrapped function will perform a simple integer factorization.

```
=> (def factorize (pfn-wrap ["int"
42039483204432094832932317913181713501317130511] "Function[{},
FactorInteger[int]]"))
#'user/factorize
```

As before, we will send-off each agent with an instruction to compute this function. We will also time the thirty computations as before.

```
=> (do
     (doseq [output-agent output-vector]              ;for each agent in the 
output vector,
       (send-off output-agent (fn [_] (factorize))))  ;update the agent with 
the result of the expression
     (time
       (while (some #(nil? @%) output-vector)         ;while any of the agents 
contains a nil,
         (Thread/sleep 50))))                         ;wait for 50 more msecs
"Elapsed time: 29992.049479 msecs"
nil
```

Thirty seconds is approximately the same length of time as the task required using the parallel evaluator directly. Parallelism, then, is effective here, too.

### Wrapping Functions Automatically When Parsing

The parser knows how to wrap first class functions automatically when it encouters them. To be able to do this, the parser needs to have been called with an optional second argument containing a function wrapper.

```
=> (def pfn-wrap (get-fn-wrapper pevaluate))
#'user/pfn-wrap

=> (def pparse (get-parser kernel-link pfn-wrap))
#'user/pparse

=> (def pmath (comp pparse pevaluate))
#'user/pmath
```

Now the parser will automatically create wrapped functions wherever it encounters first-class *Mathematica* functions (i.e. functions defined with Function[] or &, not the pattern-matching syntax).

```
=> (def increment (pmath [] "Function[{x}, x + 1]"))
#'user/increment

=> increment
#<fn_wrap$fn__478$fn__32__auto____486$wrapped_fn__488 
clojuratica.fn_wrap$fn__478$fn__32__auto____486$wrapped_fn__488@1f1235b>

=> (increment 10)
11
```

The parser will wrap functions found in data structures such as lists.

```
=> (def fn-vector (pmath :vectors [] "{#+1&, #-1&, #^5&, {{#, 0}, {0, #}}&}"))
#'user/fn-vector

=> ((fn-vector 0) 2)
3

=> ((fn-vector 1) 2)
1

=> ((fn-vector 2) 2)
32

=> ((fn-vector 3) 2)
((2 0) (0 2))
```

Because we created our parser using pfn-wrap (a parallel function wrapper created using a parallel evaluator), any functions created by the parser will automatically be evaluated in parallel when called from different threads. This is very cool.

# The *Mathematica* Package

## Getting Started

### Context

The *Mathematica* package provides features for calling Java classes written in Clojure. It supports a paradigm wherein the control-flow of your program is written on the *Mathematica* side, with calls to Clojure where necessary.

Calls to Clojure must take the form of calls to the methods (static methods or instance methods) of Java classes generated using Clojure's gen-class functionality. See the Clojure's guide on compilation and class-generation for more information on how gen-class works.

Clojuratica does not support the runtime execution of arbitrary Clojure expressions (as if a REPL were embedded in *Mathematica*). This is for both technical and performance reasons. Even if there were no technical difficulties with the execution of arbitrary Clojure expressions, method calls to compiled classes would be preferable for performance reasons alone.

Veteran *Mathematica* users will note that Clojuratica's interface to Clojure is similar to *Mathematica*'s interface to Java. They are both based on calls to the methods of compiled classes. Indeed, the Clojure interface piggy-backs on *Mathematica*'s existing Java interface, and the syntax is nearly identical. Clojuratica's contribution is merely to ensure that the data types of method arguments and return values are translated transparantly.

The following sections illustrate how to use Clojuratica's special syntax for calling Clojure-generated Java classes.

### Basic Setup

First, make sure that you have followed the installation instructures for both the Clojure and *Mathematica* components.

Now we can get started. Tell *Mathematica* to load the Clojuratica package.

```
<< "Clojuratica`"
```

This command will succeed silently if *Mathematica* is able to find and load the package.

## Calling Clojure

### An Illustrative Clojure Class

To illustrate Clojuratica's features for accessing Clojure, we will create and compile a simple Java class using Clojure. Copy the following code into a .clj file named SimpleClass.clj and compile it according to the instructions in Clojure's guide on compilation and class-generation.

```
(ns clojuratica.test.SimpleClass
  (:gen-class
     :methods [#^{:static true} [increment          [Object]
Object]
                                [decrement          [Object]
Object]
               #^{:static true} [pair               [Object Object]
Object]
```

```
               #^{:static true} [myMap              [Object Object]
Object]
               #^{:static true} [mmaIncrement       [Object Object Object]
Object]]))

(defn -increment [obj]
  (inc obj))

(defn -decrement [this obj]
  (dec obj))

(defn -pair [coll1 coll2]
  (partition 2 (interleave coll1 coll2)))

(defn -myMap [f coll]
  (map f coll))

(defn -mmaIncrement [obj evaluate parse]
  (parse (evaluate ["obj" obj] "obj + 1")))
```

The class has four static methods and one instance method.

Note that the class is written just as if its methods were going to be called from Clojure. It is no different from an ordinary Clojure class.

The decrement method is an instance method for illustration purposes. When calling Java classes from *Mathematica* there is a performance advantage to using static methods instead of instance methods. In cases where you don't need object state you should use static methods. Static methods are more coherent with Clojure's functional paradigm anyway.

In some cases, however, you may wish to create stateful objects and instantiate a number of them in *Mathematica*. *Mathematica* and Clojuratica support calls to both static and instance methods.

Place the compiled .class or .jar files in a location where *Mathematica*'s classloader can find them. A location anywhere in your system's Java classpath should work. See the *Mathematica J/Link* documentation for more detailed information on how *Mathematica* locates Java classes.

After the class is compiled and copied to the Java classpath, we can load it in *Mathematica*.

```
LoadJavaClass["clojuratica.test.SimpleClass"]
```

```
JavaClass[clojuratica.test.SimpleClass, <>]
```

We now have access to any of the class's static methods.

### Calling Static Methods

If you are familiar with *Mathematica*'s *J/Link* interface, you know that the syntax for calling static Java methods is straightforward.

```
ClassName`MethodName[arguments]                (* Mathematica call to Java *)
```

The syntax for calling the methods of Clojure-compiled classes is exactly the same, except that the call is embedded in a call to the Clojure function. The Clojure function is a piece of syntactic sugar Clojuratica provides.

```
Clojure[ClassName`MethodName[arguments]]       (* Clojuratica call to Clojure *)
```

The preferred convention is to use *Mathematica*'s prefix notation (@) syntax:

```
Clojure@ClassName`MethodName[arguments]        (* canonical syntax *)
```

Methods called in this way will have their arguments automatically converted to Clojure data types before the method call and their return value automatically parsed into a *Mathematica* data type after the method call.

Let's try it with the increment method of SimpleClass.

```
Clojure@SimpleClass`increment[1]
```

```
2
```

It's as easy as that.

The pair method of SimpleClass takes two Clojure collections and returns a sequence of their elements paired up. Clojuratica automatically converts the data types of arguments, so we can easily call the method with *Mathematica* lists.

```
Clojure@SimpleClass`pair[{1, 2, 3}, {4, 5, 6}]
```

```
{{1, 4}, {2, 5}, {3, 6}}
```

### Calling Instance Methods

The syntax for calling instance methods is almost identical to that for calling static methods. If you are familiar with *Mathematica*'s *J/Link* interface, you know that the syntax for calling instance methods in Java.

```
InstanceName@MethodName[arguments]             (* Mathematica call to Java *)
```

With Clojuratica, just enclose it in a call to Clojure.

```
Clojure[InstanceName@MethodName[arguments]]    (* Clojuratica call to Clojure *)
```

Again, the preferred convention is to use *Mathematica*'s prefix notation (@) syntax:

```
Clojure@InstanceName@MethodName[arguments]     (* canonical syntax *)
```

In addition, you can preface calls to JavaNew[] with Clojure@ to perform automatic translation of all constructor arguments. (Note that in this case the return value won't have any conversion done to it. You want the return value to be a Java object reference!)

```
Clojure@JavaNew["classname", arguments]
```

Methods called in this way will have their arguments automatically converted to Clojure data types before the method call and their return value automatically parsed into a *Mathematica* data type after the method call.

Let's try it with SimpleClass.

```
instance = Clojure@JavaNew["clojuratica.test.SimpleClass"];
Clojure@instance@decrement[10]
```

```
9
```

### Passing *Mathematica* Functions to Clojure

By default, if you try to pass a first-class *Mathematica* function to Clojure, the function will be received in Clojure as an inert CExpr object.

We can illustrate using the myMap method of SimpleClass. The method expects to be passed a function and a collection. It maps the function across the collection using Clojure's map, and returns the result.

If in *Mathematica* we attempt to pass MyMap a first-class function we get an error.

```
Clojure@SimpleClass`myMap[Function[{x}, x + 1], {1, 2, 3}]
 (* produces an error *)
```

```
Java::excptn :
  A Java exception occurred: java.lang.RuntimeException: java.lang.ClassCastException: clojuratica.CExpr
          cannot be cast to clojure.lang.IFn
    at clojure.lang.LazySeq.seq(LazySeq.java:46)
    at clojure.lang.LazySeq.hashCode(LazySeq.java:93)
Caused by: java.lang.ClassCastException:
          clojuratica.CExpr cannot be cast to clojure.lang.IFn
    at clojure.core$map__3815$fn__3817.invoke(core.clj:1503)
    at clojure.lang.LazySeq.seq(LazySeq.java:41)
    ... 10 more.

Java::argxs1 :
  The static method clojuratica`CHelper`parse was called with an incorrect number or type of arguments.
          The argument was $Failed.
```

```
$Failed
```

But if we tell it to, Clojuratica will automatically convert first-class *Mathematica* functions to Clojure functions. Simply set the Clojuratica`fnWrap global variable to True.

```
Clojuratica`fnWrap = True
```

```
True
```

Now first-class *Mathematica* functions passed as arguments will be converted to first-class Clojure functions automatically.

```
Clojure@SimpleClass`myMap[Function[{x}, x + 1], {1, 2, 3}]
```

```
{2, 3, 4}
```

If and when a function converted in this way is called on the Clojure side, the default behavior is to evaluate the function call using a serial evaluator connected to the *Mathematica* kernel that initiated the method call. This is perfectly fine behavior in most cases and doesn't require any additional setup.

If instead you want the converted functions to use a different kernel, or use a parallel evaluator, you will have to create a new evaluator yourself and tell Clojuratica where it is located. To do this, use the Clojuratica`evaluator global variable.

Let's try this. Create a new KernelLink and a new parallel evaluator attached to it. We'll store the parallel evaluator in the variable Clojuratica`evaluator. The first three commands below are generic *J/Link* commands to create a new KernelLink. The last two create a parallel evaluator using the clojuratica.CLink class (which provides Java access to the Clojuratica function generators).

```
LoadJavaClass["com.wolfram.jlink.MathLinkFactory"];
kl = MathLinkFactory`createKernelLink[
   "-linkmode launch -linkname 'c:\\\\program files\\\\wolfram
     research\\\\mathematica\\\\7.0\\\\mathkernel.exe'"];
kl@discardAnswer[];
LoadJavaClass["clojuratica.CLink"];
Clojuratica`evaluator = CLink`getEvaluator[kl, ":parallel"]
```

```
« JavaObject[clojuratica.parallel_evaluator$get _evaluator __
   339 $flag_parser __ 32 __auto ___ _ 348 $parallel_evaluator __ 354] »
```

Note that flags *must* be passed in a (possibly empty) string as the final argument to CLink`getEvaluator[].

From this point on, any first-class *Mathematica* functions passed as method arguments in a Clojure[] call will be converted to first-class Clojure functions that, when called, are evaluated using a parallel evaluator attached to the newly created kernel.

```
Clojure@SimpleClass`myMap[Function[{x}, x + 1], {1, 2, 3}]
 (* wrapped function will be evaluated in the new kernel *)
```

```
{2, 3, 4}
```

You can kill the new kernel with:

```
kl@terminateKernel[]
```

To restore the default evaluator (a serial evaluator connected to the *Mathematica* kernel making the method calls) do the following.

```
LoadJavaClass["com.wolfram.jlink.StdLink"];
Clojuratica`evaluator = CLink`getEvaluator[StdLink`getLink[], ":serial"]
```

```
« JavaObject[clojuratica.serial_evaluator$get _evaluator
   __ 202 $flag_parser __ 32 __auto ___ _ 210 $serial_evaluator __ 212] »
```

Warning: You must not try to replace :serial with :parallel here. There is no way to create a *parallel* evaluator connected to the *Mathematica* kernel making the method calls. Any calls to such an evaluator will cause *Mathematica* to freeze up horribly. If you want automatic function-wrapping of method arguments, and you want to have those functions evaluated on a parallel evaluator, you must launch a separate kernel using MathLinkFactory`createKernelLink[] and tie an evaluator to it using CLink`getEvaluator[].

### Calling Clojure Methods that then Evaluate *Mathematica* Expressions

If you want your Clojure methods to themselves be able to interact with *Mathematica*, you can create any of the the Clojuratica functions (the evaluator, the parser, etc.) in *Mathematica* and pass them to your Clojure methods. The last section hinted at how you might do this. Here is some more detail.

Here is how you might create an evaluator and parser that use the active *Mathematica* kernel.

```
LoadJavaClass["com.wolfram.jlink.StdLink"];
LoadJavaClass["clojuratica.CLink"];
evaluate = CLink`getEvaluator[StdLink`getLink[], ":serial"]
 (* first argument is a KernelLink,
second argument is a string containing flags *)
parse = CLink`getParser[StdLink`getLink[], Null, ""]
 (* first argument is a KernelLink, second argument is a function
 wrapper or Null third argument is a string containing flags *)
```

```
« JavaObject[clojuratica.serial_evaluator$get _evaluator
   __ 202 $flag_parser __ 32 __auto ___ _ 210 $serial_evaluator __ 212] »
```

```
« JavaObject[clojuratica.clojuratica$get_parser
   __ 550 $flag_parser __ 32 __auto ___ _ 559 $parser__ 562] »
```

You could then pass these functions to a Clojure method that will use them to call back into *Mathematica.* The mmaIncrement method of SimpleClass, for instance, expects to be passed an evaluator and parser. It returns its first argument plus one.

```
Clojure@SimpleClass`mmaIncrement[8, evaluate, parse]
```

```
9
```

Instead of tying your methods back into the active kernel, you could instead create a completely new kernel using MathLinkFactory`createKernelLink[] and use that kernel to create an evaluator. We did this in the previous section, and in general it is the better way of doing things. By creating an altogether new kernel you gain the option of using Clojuratica's parallel evaluator. (You should not try to create a parallel evaluator attached to the active kernel. Things will go badly.)

```
LoadJavaClass["com.wolfram.jlink.MathLinkFactory"];
kl@terminateKernel[];
(* unload the kernel we launched in the previous section *)
kl = MathLinkFactory`createKernelLink[
   "-linkmode launch -linkname 'c:\\\\program files\\\\wolfram
     research\\\\mathematica\\\\7.0\\\\mathkernel.exe'"];
kl@discardAnswer[];
LoadJavaClass["clojuratica.CLink"];
evaluate = CLink`getEvaluator[kl, ":parallel"]
 (* first argument is a KernelLink,
second argument is a string containing flags *)
parse = CLink`getParser[kl, Null, ""]
 (* first argument is a KernelLink, second argument is a function
 wrapper or Null third argument is a string containing flags *)
```

```
« JavaObject[clojuratica.parallel_evaluator$get _evaluator __
   339 $flag_parser __ 32 __auto ___ _ 348 $parallel_evaluator __ 354] »
```

```
« JavaObject[clojuratica.clojuratica$get_parser
   __ 550 $flag_parser __ 32 __auto ___ _ 559 $parser__ 562] »
```

The following uses the new parallel evaluator running on the newly created kernel.

```
Clojure@SimpleClass`mmaIncrement[8, evaluate, parse]
```

```
9
```

That covers the features of Clojuratica. Enjoy!