A beginner-friendly environment for exploring error messages in the Clojure programming language.

Tristan Kalvoda, Elena Machkasova, Jaydon Stanislowski, and John Walbran

University of Minnesota, Morris

Midwest Instruction and Computing Symposium, April 2025

Outline

Clojure Language and Syntax

What is Clojure?

- Clojure is a part of the Lisp language family
  - prefix notation (operators before operands)
  - expressions are surrounded by parentheses

  Example: `(/ 9 3)` denotes 9 divided by 3
- Clojure is an interpreted language
- It's implemented in Java and runs on the Java Virtual Machine (JVM) interpreter

Clojure code $\rightarrow$ Java code $\rightarrow$ JVM bytecode $\rightarrow$ executed on JVM

Clojure Language and Syntax

Clojure's REPL

- Interactive environment for code evaluation
- Read → Evaluate → Print → Loop (REPL)

```
Clojure 1.11.4
user=> (count [-1 0 1])
3
user=>
```

Clojure's Error Messages

Clojure Exceptions

- All exceptions in Clojure are Java exceptions
- Clojure syntax errors will also result in an exception

Error Messages

- Generate when a exception occurs
- Provide error type, cause, and location

Clojure's Error Messages

Anatomy of a Clojure Error Message

```
=> (/ 9 0)
Execution error (ArithmeticException) at user/eval1
(REPL:1).
Divide by zero
```

- `ArithmeticException`: The type of error that occurred
- `user/eval1 (REPL:1)`: The location where the error happened (in this case, REPL, line 1)
- `Divide by zero`: The description of the error's cause

Motivation

- UMN Morris has been teaching functional languages first (Scheme, Racket) to students for over 30 years
- Historically done at other major institutions, including MIT
- This practice has many pedagogical benefits
- Clojure is good to learn due to its popularity and widespread industry use, but has flaws in its presentation of error messages

Goals of Babel

- Designed to be an interactive, beginner-friendly tool for understanding error data in Clojure
- Simplifies error messages to be more intuitive, removing jargon and clutter
- Utilizes tools to make Clojure errors more descriptive and detailed for formatting messages

Current State of Babel

- Our group began working after error replacement was largely achieved
- Present goals include exploring more interactive, IDE-like tools that integrate with Babel
- Need the capacity to support testing and usability
- Lots of refactoring work is being done to support this, as well as refining Babel itself

Motivation and Goals

Example

Consider the error produced by the form below. What does it mean?

```
=> (count 1)
Execution error (UnsupportedOperationException) at
user/eval1529 (REPL:1).
count not supported on this type:  Long
```

On Clojure *spec*

Babel is based on an existing Clojure library called *spec*:

- Allows specifying requirements on arguments of functions
- Requirements are predicates: can check argument types, count, values, etc.
- We bind specs to core Clojure functions
- If requirements aren't met then a *spec* error happens
- Error reports from spec are more detailed than native Clojure error messages

Overview

Babel tool:

- Replaces native Clojure error messages
- Messages produced by Babel broadly fall into two types:

Spec errors:

- Babel uses *spec* to specify conditions on Clojure functions
- *spec* provides information to make error messages more precise

Non-spec errors:

- When *spec* cannot be provided, e.g. syntax errors, regex is used to identify error messages
- We use a dictionary to rephrase these errors

Usage

- Launching a REPL server in the Babel repository allows the tool to "hook" to it
- Babel can report errors on a loaded Clojure file
- Babel catches Clojure errors, including our *spec* reports
- Error messages are replaced with modified ones
- Currently only supports REPL, not IDE

Setup

```
user=> (count 1)
Execution error (UnsupportedOperationException) at user/eval1531 (REPL:1).
count not supported on this type: Long
```

Figure: A default error on the `(count)` function, caused by a mistake that a student typically makes.

Setup

```
babel.middleware=> (count 1)
Function count does not allow a number as an argument in this position.

In Clojure interactive session on line 1.
Call sequence:
[Clojure interactive session (repl)]
```

Figure: The error message on the same form produced by Babel, containing a description of the potential problem in plain English.

Morse

- We would like to expand error message handling from REPL strings to interactive visual environments

Morse

- We would like to expand error message handling from REPL strings to interactive visual environments
- Joe Lane (Nubank) suggested a new tool called *Morse*

Morse

- We would like to expand error message handling from REPL strings to interactive visual environments
- Joe Lane (Nubank) suggested a new tool called *Morse*
- Morse is a third party data visualization and navigation tool

Morse

- We would like to expand error message handling from REPL strings to interactive visual environments
- Joe Lane (Nubank) suggested a new tool called *Morse*
- Morse is a third party data visualization and navigation tool
- Morse provides a customizable set of viewers for different data structures, like text, maps, or HTML content

Morse

- We would like to expand error message handling from REPL strings to interactive visual environments
- Joe Lane (Nubank) suggested a new tool called *Morse*
- Morse is a third party data visualization and navigation tool
- Morse provides a customizable set of viewers for different data structures, like text, maps, or HTML content
- We are developing a custom set of viewers for error messages

Morse

- We would like to expand error message handling from REPL strings to interactive visual environments
- Joe Lane (Nubank) suggested a new tool called *Morse*
- Morse is a third party data visualization and navigation tool
- Morse provides a customizable set of viewers for different data structures, like text, maps, or HTML content
- We are developing a custom set of viewers for error messages
- We need to label different parts of error message for different viewers

Morse Default

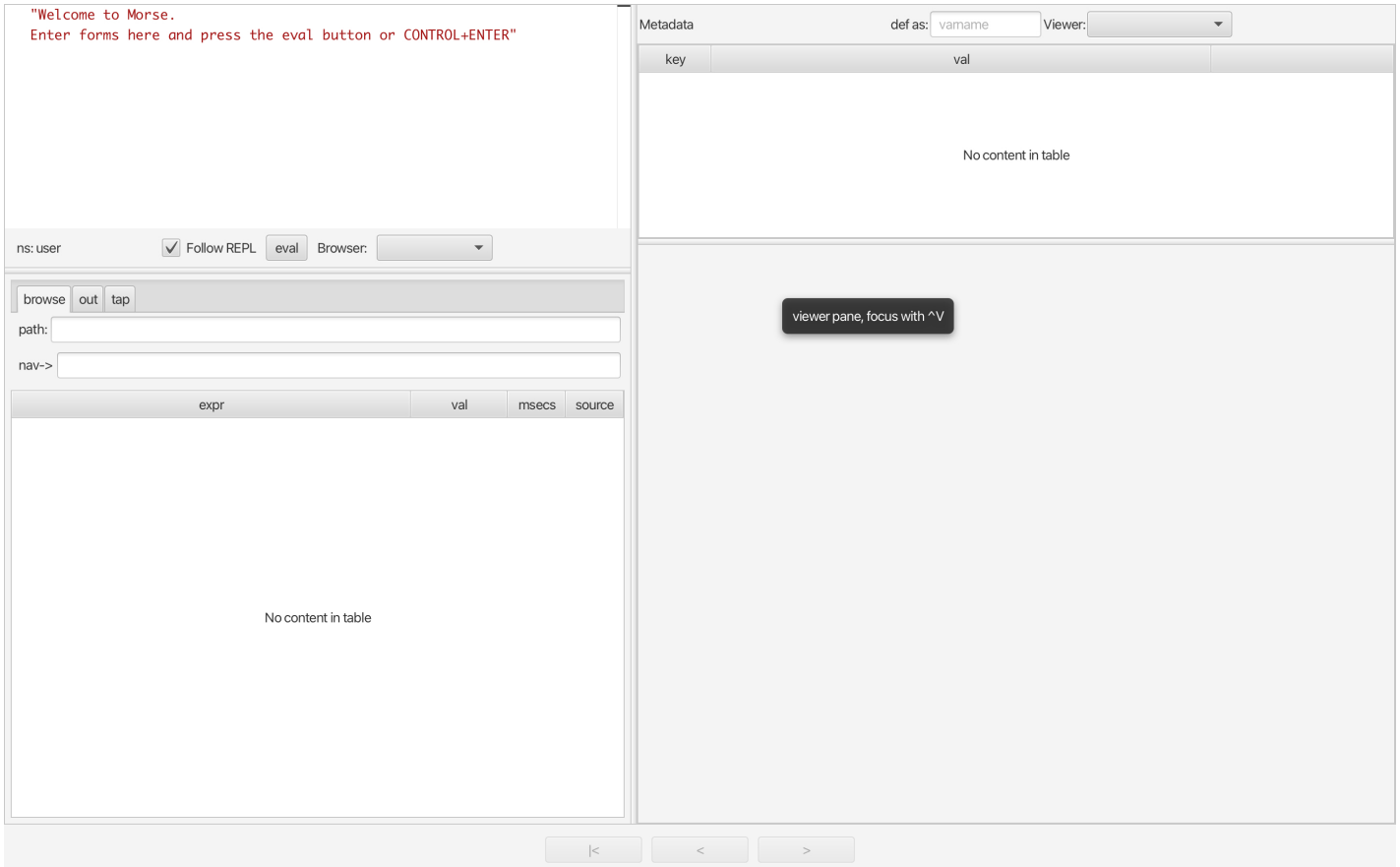

Figure: The default view of the Morse tool.

Morse Lists

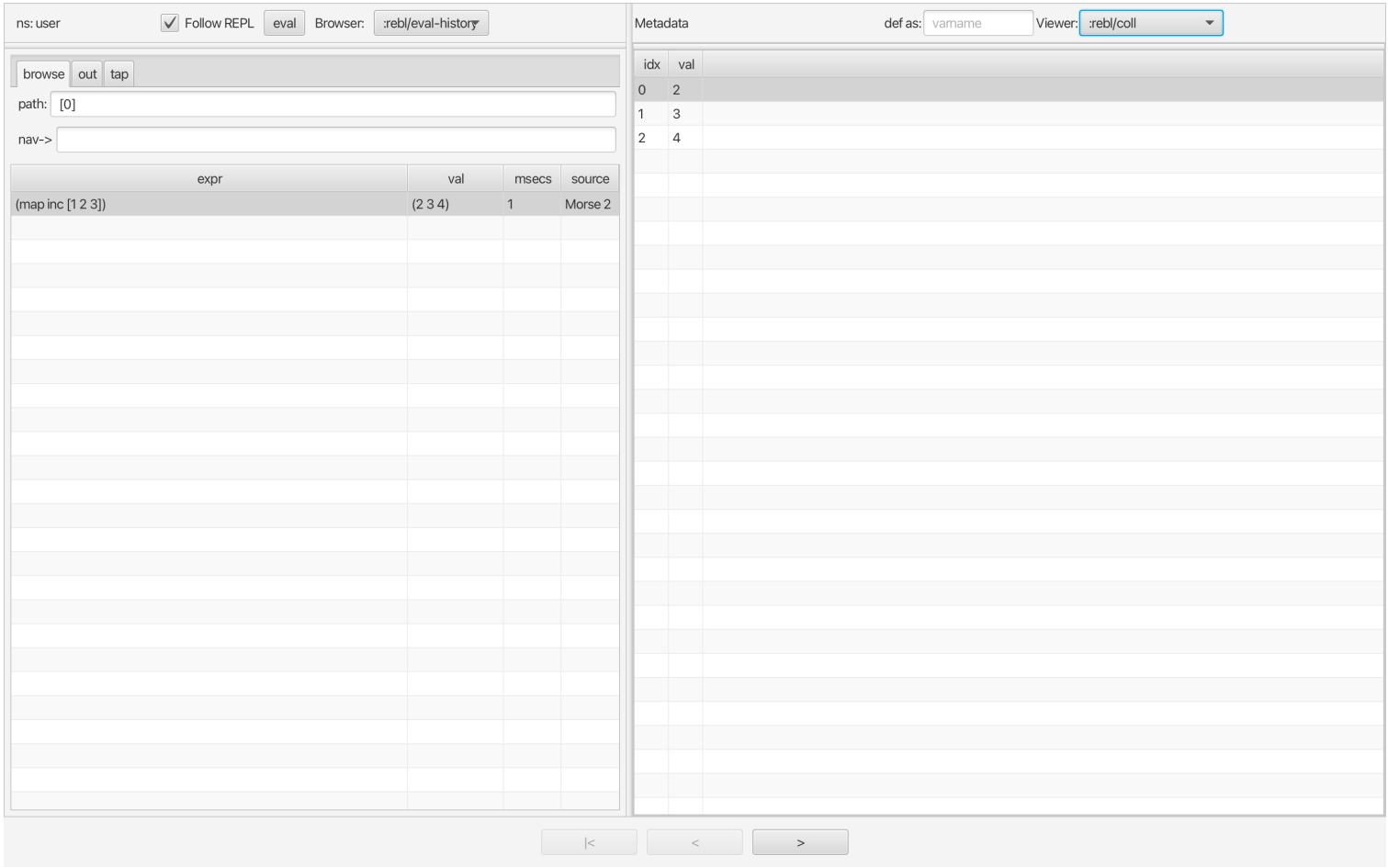

Figure: Morse viewer for vector data. Shows the result of the form `(map inc [1 2 3])`.

Integrating with Morse

- In order to connect to a Morse session, we introduce a new session layer that exposes communication hooks for other tools.

Integrating with Morse

- In order to connect to a Morse session, we introduce a new session layer that exposes communication hooks for other tools.
- `:init` Behavior on startup, launches a new Morse session connected to the current REPL.

Integrating with Morse

- In order to connect to a Morse session, we introduce a new session layer that exposes communication hooks for other tools.
- `:init` Behavior on startup, launches a new Morse session connected to the current REPL.
- `:eval` Behavior on form evaluation. Stores the command and sends it to REPL and Morse.

Integrating with Morse

- In order to connect to a Morse session, we introduce a new session layer that exposes communication hooks for other tools.
- `:init` Behavior on startup, launches a new Morse session connected to the current REPL.
- `:eval` Behavior on form evaluation. Stores the command and sends it to REPL and Morse.
- `:caught` Behavior on caught exception. Processes the error in Babel, and changes the Exception←:wantString processing to instead create a vector of labelled pairs.

Our Morse Viewer

- We used the existing WebView structure within Morse to render error messages.

Our Morse Viewer

- We used the existing WebView structure within Morse to render error messages.
- We use HTML to add color coding for important terms, and mono-space fonts to highlight code chunks within the message.

Our Morse Viewer

- We used the existing WebView structure within Morse to render error messages.
- We use HTML to add color coding for important terms, and mono-space fonts to highlight code chunks within the message.
- The exact design of the final viewers are still a work in progress.

```
user=> (even? 1 2)
Execution error (ArityException) at user/eval2044 (REPL:1).
Wrong number of args (2) passed to: clojure.core/even?
```

Figure: The output for the form `(even? 1 2)` in default Clojure.

```
babel.middleware=> (even? 1 2)
Wrong number of arguments in (even? 1 2): the function even? expects one
argument but was given two arguments.
In Clojure interactive session on line 1.
Call sequence:
[Clojure interactive session (repl)]
```

Figure: The output for the form `(even? 1 2)` in Babel through the REPL.

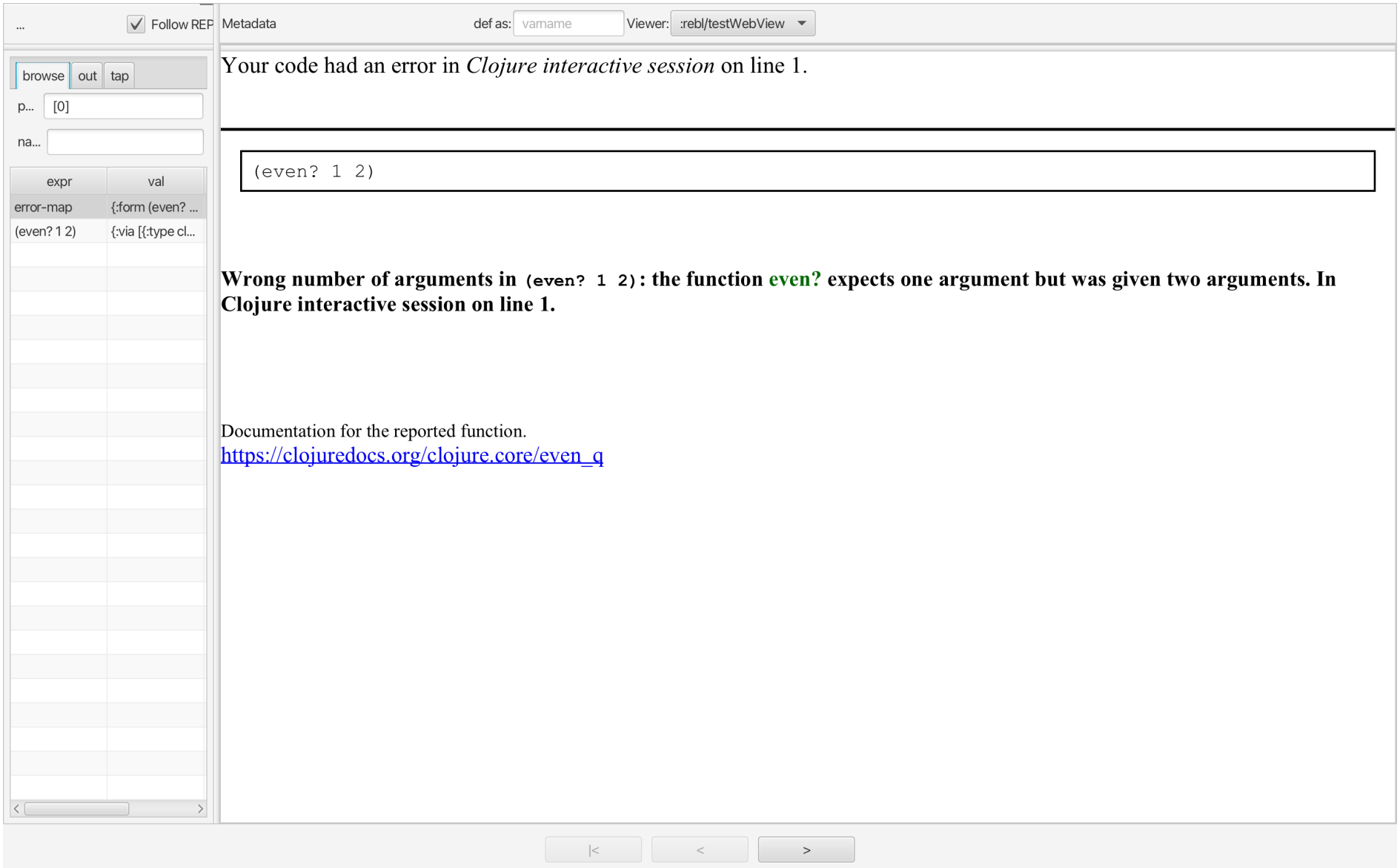

Figure: The output for the form `(even? 1 2)` in Babel with Morse.

Current Work

Work in Progress

- This year's work has involved exploring interactions between Babel and Morse
- Babel outputs only strings to REPL, Morse needs labeled data to apply viewers
- We are focusing on information flow between REPL and Morse
- We are also prototyping models for Morse viewers

Future Work

We want to:

- finalize data labeling and corresponding viewers
- explore interactive capabilities of Morse, e.g. hover text to clarify terms, hyperlinks to documentation
- develop Morse viewers for other information, such as the stack trace, and full Java error messages
- conduct usability studies with our developments
- explore IDE (VS Code) integration for workflow refinements

Acknowledgements

This work was supported in part by Morris Academic Partnership (MAP) and UMN Undergraduate Research Opportunity (UROP).

We thank Joe Lane for introducing us to Morse tools and for numerous helpful discussions.

Discussion

Questions?