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 language and Syntax

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

Example: `(/ 9 3)` denotes 9 divided by 3

Clojure Language and Syntax

- Clojure `Elena: is implemented in Java and` runs on the Java Virtual Machine (JVM)
  - executed code compiles to JVM bytecode `Elena: I corrected the line below slightly. Not sure if you need this line.`
- Clojure code → Java code → JVM bytecode → executed on JVM

Clojure Language and Syntax

Clojure's REPL

- interactive environment for code evaluation
- Read → Evaluate → Print → Loop `Tristan: repl example image`

Clojure's Error Messages

Clojure Exceptions

- an event or error that disrupts the normal flow of a program's execution
- Clojure syntax errors will also result in an exception `Elena: mention that it is a Java exception`

Error Messages

- generate when a exception occurs
- provide error type 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.

Clojure's Error Messages

Exception Example

```
#error {
:cause "Divide by zero"
:via
[:type java.lang.ArithmeticException
:message "Divide by zero"
:at [clojure.lang.Numbers divide "Numbers.java"
190]}]
:trace
[[clojure.lang.Numbers divide "Numbers.java" 190]
...  omitting 18 lines...
[clojure.main main "main.java" 40]] Tristan:  image
instead?  or maybe not add this Elena:  I don't think
you need this slide
```

Setup and Goals

Overview of Babel

- Tool designed to replace native Clojure messages to aid in understanding
- Relies heavily on the Clojure spec library to catch errors on function calls `Elena: We didn't introduce spec yet - can introduce it here; show "spec" and "other errors" in boldface or some such.`
- Maintains a dictionary of other errors (e.g. division by zero) that can't be spec'd, in order to rewrite them as well `Elena: Using RegEx to pull out different parts`

Usage

- Launching a REPL server in the Babel repository allows the tool to "hook" to it
- Initialization function `(setup-exc)` is called to begin intercepting error messages `Elena: I don't think we need to mention this`

Setup and Goals

Motivation `Elena: shouldn't this be before the previous slide?`

- Babel is a learning tool for beginners to Clojure and programming as a whole
- Clojure error messages contain awkward phrasing that may impede understanding

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
```

Setup and Goals

Exceptions Processing

Sending Data to Morse

- The Clojure REPL does not provide the proper hooks to effectively manipulate error message data.
- To get around this, we need to initialize Babel within a sub-REPL of the parent REPL session.
- Creating a sub-REPL allows us to introduce hooks that let us add preprocessing steps.

Sending Data to Morse

- The Clojure REPL does not provide the proper hooks to effectively manipulate error message data.
- To get around this, we need to initialize Babel within a sub-REPL of the parent REPL session.
- Creating a sub-REPL allows us to introduce hooks that let us add preprocessing steps.

Sending Data to Morse

- The Clojure REPL does not provide the proper hooks to effectively manipulate error message data.
- To get around this, we need to initialize Babel within a sub-REPL of the parent REPL session.
- Creating a sub-REPL allows us to introduce hooks that let us add preprocessing steps.

Sub-REPL hooks

Babel uses the following hooks as part of error processing:

- `:init` Defines initial behavior on creation. In Babel this starts a new Morse session connected to the current REPL.
- `:eval` Defines behavior when a command is run. In Babel this stores the command verbatim into an atom, and evaluates the command in both the REPL and Morse.
- `:caught` Defines behavior on an exception. In Babel this processes the error, and passes the following information to Morse for display in a custom viewer:
  - The last command entered, read from an atom that is updated at evaluation.
  - The location in the environment where the error occurred. In the REPL, this resolves to "Clojure Interactive Session".
  - A vector of pairs containing the error message produced by Babel, with labels associated with each segment denoting its type for formatting.
  - The url to the documentation of the function called that

Sub-REPL hooks

Babel uses the following hooks as part of error processing:

- `:init` Defines initial behavior on creation. In Babel this starts a new Morse session connected to the current REPL.
- `:eval` Defines behavior when a command is run. In Babel this stores the command verbatim into an atom, and evaluates the command in both the REPL and Morse.
- `:caught` Defines behavior on an exception. In Babel this processes the error, and passes the following information to Morse for display in a custom viewer:
  - The last command entered, read from an atom that is updated at evaluation.
  - The location in the environment where the error occurred. In the REPL, this resolves to "Clojure Interactive Session".
  - A vector of pairs containing the error message produced by Babel, with labels associated with each segment denoting its type for formatting.
  - The url to the documentation of the function called that

Sub-REPL hooks

Babel uses the following hooks as part of error processing:

- `:init` Defines initial behavior on creation. In Babel this starts a new Morse session connected to the current REPL.
- `:eval` Defines behavior when a command is run. In Babel this stores the command verbatim into an atom, and evaluates the command in both the REPL and Morse.
- `:caught` Defines behavior on an exception. In Babel this processes the error, and passes the following information to Morse for display in a custom viewer:
  - The last command entered, read from an atom that is updated at evaluation.
  - The location in the environment where the error occurred. In the REPL, this resolves to "Clojure Interactive Session".
  - A vector of pairs containing the error message produced by Babel, with labels associated with each segment denoting its type for formatting.
  - The url to the documentation of the function called that

Current State of the Project

- We have existing error messages without labels for many common errors of core functions.
- We can connect Morse to a REPL session, and have mirroring form evaluation.
- Most of the work this year was spent structuring things for integration with Morse viewers.
- The introduction of the error labeling and prototyping this was pivotal in enabling data formatting.
- We currently have a small number of error messages labeled for demonstration purposes.

Current State of the Project

- We have existing error messages without labels for many common errors of core functions.
- We can connect Morse to a REPL session, and have mirroring form evaluation.
- Most of the work this year was spent structuring things for integration with Morse viewers.
- The introduction of the error labeling and prototyping this was pivotal in enabling data formatting.
- We currently have a small number of error messages labeled for demonstration purposes.

Current State of the Project

- We have existing error messages without labels for many common errors of core functions.
- We can connect Morse to a REPL session, and have mirroring form evaluation.
- Most of the work this year was spent structuring things for integration with Morse viewers.
- The introduction of the error labeling and prototyping this was pivotal in enabling data formatting.
- We currently have a small number of error messages labeled for demonstration purposes.

Current State of the Project

- We have existing error messages without labels for many common errors of core functions.
- We can connect Morse to a REPL session, and have mirroring form evaluation.
- Most of the work this year was spent structuring things for integration with Morse viewers.
- The introduction of the error labeling and prototyping this was pivotal in enabling data formatting.
- We currently have a small number of error messages labeled for demonstration purposes.

Current State of the Project

- We have existing error messages without labels for many common errors of core functions.
- We can connect Morse to a REPL session, and have mirroring form evaluation.
- Most of the work this year was spent structuring things for integration with Morse viewers.
- The introduction of the error labeling and prototyping this was pivotal in enabling data formatting.
- We currently have a small number of error messages labeled for demonstration purposes.

Future Work

The following are areas of active development:

- Expand data labeling to all Babel error messages.
- Add hover text for specific terms to add definitions and supplementary information to the presented error message.
- Refining the end user work flow between working code and erroring code.
- Develop Morse viewers for other information, such as the stack trace, and full java error messages.

Future Work

The following are areas of active development:

- Expand data labeling to all Babel error messages.
- Add hover text for specific terms to add definitions and supplementary information to the presented error message.
- Refining the end user work flow between working code and erroring code.
- Develop Morse viewers for other information, such as the stack trace, and full java error messages.

Future Work

The following are areas of active development:

- Expand data labeling to all Babel error messages.
- Add hover text for specific terms to add definitions and supplementary information to the presented error message.
- Refining the end user work flow between working code and erroring code.
- Develop Morse viewers for other information, such as the stack trace, and full java error messages.

Future Work

The following are areas of active development:

- Expand data labeling to all Babel error messages.
- Add hover text for specific terms to add definitions and supplementary information to the presented error message.
- Refining the end user work flow between working code and erroring code.
- Develop Morse viewers for other information, such as the stack trace, and full java error messages.

Future Work (cont.)

`Elena: simplify the sentences`

- Once we have greater feature coverage in Babel, we plan to run a usability study about the interactive tools we have developed.
- We are going to use the results of letting users explore our tools while learning Clojure in order to guide further design.
- We would like to explore IDE integration to further expand possible work-flow refinements.

Future Work (cont.)

`Elena: simplify the sentences`

- Once we have greater feature coverage in Babel, we plan to run a usability study about the interactive tools we have developed.
- We are going to use the results of letting users explore our tools while learning Clojure in order to guide further design.
- We would like to explore IDE integration to further expand possible work-flow refinements.

Future Work (cont.)

`Elena: simplify the sentences`

- Once we have greater feature coverage in Babel, we plan to run a usability study about the interactive tools we have developed.
- We are going to use the results of letting users explore our tools while learning Clojure in order to guide further design.
- We would like to explore IDE integration to further expand possible work-flow 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?