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

Tristan Kalvoda, Elena Machkasova, Jaydon Stanislowski, and John Walbran
Computer Science Discipline
University of Minnesota Morris
Morris, MN 56267
kalvo007@umn.edu, elenam@umn.edu, stani152@umn.edu, walbr042@umn.edu

## Abstract

The Clojure programming language, built on the Java Virtual Machine, is a widely used functional language. Its simple syntax makes it a powerful language to introduce beginner programmers to key concepts of programming. However, debugging in Clojure can be unintuitive because Clojure errors are exceptions in the underlying Java language. They frequently contain terminology intended for experienced developers, leaving out important context. Our research project, Babel, aims to provide beginner programmers with a tool that replaces Clojure error messages with clearer explanations, removing unfamiliar jargon. This is work in progress, with the current goal of improving the user experience by using Morse, a third-party viewer, to present error information in a more beginner-friendly format. In this paper we discuss the structure of Clojure error messages and explain the data flow for the error messages processing. We discuss options for formatting the error messages provided by Morse, as well as the principles that guide our choices. We present examples of views and interactions for Clojure error messages. We conclude with a discussion of future directions.

# 1 Introduction

The Clojure programming language is a functional language in the Lisp family, developed and released in 2007 by Rich Hickey [7]. Clojure is implemented in Java, compiles to Java bytecode, and runs in the Java Virtual Machine (JVM), allowing programmers to use Java libraries directly from Clojure. However, unlike Java, it is a dynamically typed functional language with immutable data as a default (mutable data must be specifically declared). Clojure quickly became known in industry. Even though the number of Clojure programmers is not a large percentage of all developers (1.3% of professional developers reported extensive use of Clojure), it remains one of the most admired languages (68.5%) and is the 3rd highest-paying programming language, according to the 2024 Stack Overflow Developers Survey [2].

There is a long history of teaching Lisp as the first programming language, starting from the famous "Structure and Interpretation of Computer Programs" curriculum [3] and continuing with the introduction of the Racket programming language that provides a tiered system of Lisp-based beginner languages [5, 1]. While Clojure is also a Lisp language and shares the benefits of focusing on development of functions that follow the structure of the data representation, some of Clojure's aspects make it more challenging than other Lisps for novice programmers.

One such aspect is Clojure error messages: due to Clojure implementation via the underlying Java language, Clojure errors are Java exceptions. They frequently contain terminology intended for experienced developers, leaving out important context. As a result, what would otherwise be a powerful educational tool presents a difficult learning curve for new programmers.

A project at UMN Morris, called Babel, aims at providing alternative beginner-friendly error messages and other tools that help beginner programmers to interpret Clojure errors. The current state of the project provides a tool to rewrite the majority of error messages into a clearer and more specific form - for example, specifying the erroneous arguments to functions. The challenge at this point lies in connecting this tool to a variety of possible ways of interacting with Clojure (via an interpreter or in IDE, for example). We also are working on providing features to give the users more resources and more control when they are receiving an error message. The resources include explanation of terminology (for example, what does the term "hashmap" refer to?), links to documentation for functions, etc. Giving users more control allows for optional expansion of some information, such as the stack trace. This work is still in progress. In this paper we discuss our approaches to tackling this task, demonstrate our current achievements, and discuss future directions.

The rest of the paper is structured as following: Section 2 provides background on Clojure, its error messages, and the Clojure feature called *spec* that we utilize on our work. Section 3 discusses how Babel processes error messages. Section 4 describes the interactive environment that we are developing, both from the implementation side and from the standpoint of design for the users. Section 5 discusses the current state of the project and future directions.

### 1.1 Related Work

Previous work on the topic of improving error messages in general for beginners [4] revealed that certain visual design choices can have a strong impact on a user's comprehension of error data. Dong and Khandwala found that, out of text coloring, increased white space, and collapsible components, adding whitespace to separate important sections of the error message helped to aid with understanding the most overall, and that employing theories of visual perception in the design of error messages help to improve their usability. We are using these insights as guidance for our approach.

A 2017 paper by Wrenn and Krishnamurthi [10] on error handling in Racket, another Lisp-based programming language, suggests treating error messages as classifiers, using highlighting tools to mark up error messages. This is somewhat similar to our own approach in that we extract components of error messages and classify them using keywords to determine appropriate styling choices that make them more navigable to the end user.

An earlier paper on Racket [8] suggests that the vocabulary used in error messages plays a key role in the ability to understand them, and students often have a hard time navigating technical language intended to point them to specific issues. This knowledge is important to our own research, as native Clojure error messages often contain highly technical vocabulary as well, and we make an effort to replace such terminology with more universal language.

## 2 Background

### 2.1 Clojure and its Error Messages

Clojure syntax utilizes the prefix notation, common for Lisp languages: a Clojure expression is enclosed in parentheses that contain a function followed by its arguments. For example, `(/ 6 2)` denotes an expression $6/2$ and results in $3$. Clojure is dynamically typed, and no types are declared.

In addition to lists (common in other Lisp languages), Clojure employs a variety of collections, such as vectors, hashmaps, and sets. It comes with a large number of predefined functions. Below we introduce collections and functions used in examples in this and subsequent sections.

Vectors are enclosed in square brackets; commas between elements are optional and usually omitted. For example, `[1 -1 0]` denotes a vector of three elements: $1, -1$, and $0$. By convention Clojure uses `?` at the end of the names of predicates, i.e. functions that return a boolean. For example, `neg?` takes a number and returns *true* if it's negative and *false* otherwise. Likewise, `even?` takes an integer and returns *true* if it's even and *false* otherwise. A function `filter` is used to select elements of a vector, or another collection. It takes a predicate and a collection and returns a collection with only the elements that satisfy the predicate. For example, `(filter neg? [1 -1 0])` would return a collection with only one element, $1$. A subtle point is that Clojure also allows passing just a predicate to `filter`, without a collection: `(filter neg?)`. This returns a function

that can be invoked on a vector at some later point. Since this use is allowed, the correct number of arguments for `filter` is one or two.

Clojure is a hosted language, running on the Java Virtual Machine (JVM), which is a bytecode interpreter that allows a program to run on any computer without needing to be rewritten for each one. The JVM works by compiling code into bytecode, which it can then run. Clojure code can be loaded into the JVM from a file or directly via an interactive interpreter known as REPL, which stands for Read - Evaluate - Print - Loop. In our examples we assume that the code is entered into REPL. The REPL prompt is denoted with =>, and the Clojure response follows.

When Clojure code is executed, it compiles into JVM bytecode, allowing it to use many of the features that the JVM provides, but most importantly for the work presented here is that it uses the Java exception system for handling errors and when an exception is thrown in Clojure, it is a Java exception.

An exception is an event or error that disrupts the normal flow of a program's execution. In Clojure, exceptions typically occur when something goes wrong, such as dividing by zero or passing an inappropriate argument to a function. It's important to note that Clojure syntax errors will also result in an exception, because during the compilation of dynamically loaded Clojure code into JVM bytecode, any syntax error will cause an exception.

Error messages are generated when an exception occurs. They provide information about the type of error as well as where it happens. As an example of one such mistake, a novice to Clojure might write the following expression

```
(filter neg? 1 -1 0)
```

instead of the correct version `(filter neg? [1 -1 0])`, as described above. This causes an error, because the filter function was expecting either one or two arguments. Here 1, -1, and 0 are not in a collection, presumably because the user forgot to include them in a vector. Instead, they are being passed as separate arguments, which causes the function to be overloaded with inputs it doesn't expect.

```
Execution error (ArityException) at user/eval1 (REPL:1).
Wrong number of args (4) passed to: clojure.core/filter
```

Clojure presents this error as an *ArityException*, referring to the number of arguments of the function. The message is cryptic for beginners, especially without the documentation.

Consider another example that we use to detail representation of Clojure error messages:

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

The code is causing an error because it is trying to divide by zero, Let's break apart the error message:

- `ArithmeticException` is the type of error that happens.
- `user/eval1 (REPL:1)` is the location of the error; since for this example we assume that the code was directly typed into REPL, the location is REPL, line 1.
- `Divide by zero` is the given description for the cause of the error.

The error message is a part of a Java exception object that contains more information.

The exception object for `(/ 9 0)` contains the following information, represented as a nested Clojure hashmap, i.e. a collection of key/value pairs. The key names are preceded by `:` The exception typically contains a stack trace (see the keyword `:trace`) listing the sequence of function calls that resulted in the error. While our project includes a way of reducing the stack trace to a few informative lines, this paper is not focusing on the stack trace.

```
#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]]}
```

In addition to the above information, the exception contains the phase of Clojure execution when the exception occurred. For example, in the earlier case of the divide by zero error, the exception occurs in the execution phase. Since an error can occur at any point in the Read-Evaluate-Print-Loop, there are several possibilities of the phase. Below we list some of them:

| Phase | Description |
|---|---|
| :read-source | Error while reading characters at the REPL or from a source file. |
| :compile-syntax-check | Syntax error caught during compilation. |
| :execution | Any errors thrown at execution time. |
| :read-eval-result | Error thrown while reading the result of execution. |
| :print-eval-result | Error thrown while printing the result of execution. |

Knowing the phase of the evaluation is helpful for detecting what kind of error has occurred and what information the user would benefit from.

## 2.2 Clojure Spec

Clojure *spec* is an addition to Clojure first released around 2016 [6]. It is a set of features that allow setting specifications, particularly for functions, that are checked before the function call. These specifications typically include the number and types of arguments, although they may include other conditions. The flexible nature of spec allows programmers to choose what parameters to check and, to a degree, what information will be communicated about why the arguments failed to satisfy spec.

As an example, one may specify that a function `f` takes no fewer than 2 and no more than 4 arguments, the first argument must be a non-negative integer, and the second one must be sequence. Types (or any other conditions) are specified as predicates. For example, we can require that the first argument of `f` satisfy both `number?` and `pos?` predicates, and the second satisfies `seq?` (a predicate that determines if a Clojure object is a sequence). Any existing Clojure predicate can be used, or the user can write their own.

If arguments not satisfying the spec are passed to the function, a spec error will occur. A spec error is an exception of the type `clojure.lang.ExceptionInfo` that contains data describing the error, including the failed predicate. For example, if the second argument of the function `f` above is a number `5`, and not a sequence, the error message states that it failed the predicate `seq?` Even more importantly, the argument that's causing the error (in this case `5`) will be included in the spec error message.

Without using spec, the error message would be a Java `ClassCastException` that doesn't report the value of the failing argument, only the incompatible types. The more detailed information provided by spec allows us to make error messages more understandable to novice programmers, as described in Section 3.1.

# 3 Babel Project

## 3.1 Overview of the Project

Our research project, Babel, is a tool intended to help novice programmers understand Clojure errors. It does this by providing learners with substitute error messages that describe code problems in simpler terms, replacing the unfamiliar jargon and low-level clutter of native error messages. For example, the error message produced via the function call below, where `count` is a function that returns the length of a sequential collection:

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

is transformed into a new message, replacing the reference to the internal type `Long` and removing details about the Java exception object and irrelevant internal information:

```
=> (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)]
```

Babel creates error messages like these through simple analysis of the native error's Java exception class and message contents, and one could argue that for cases like these, the native error message suffices. However, for more complicated problems, such as incorrect calls to functions or malformed arguments, Clojure's messages get more difficult to parse. Consider an erroneous code fragment described in Section 2.1:

```
=> (filter neg? 1 -1 0)
Execution error (ArityException) at user/eval1540 (REPL
   :1).
Wrong number of args (4) passed to: clojure.core/filter
```

In these situations, it would be ideal to have a detailed explanation of what the offending arguments in the function call are, and what was expected. This is where Babel makes use of the Clojure spec library, described in Section refsubsec:spec-overview, in its error reporting.

By employing specs on core Clojure functions, it is possible to explicitly specify the correct form of a function call and how the function was used incorrectly. Using spec, we know that the `filter` function must take a predicate and, optionally, a collection, therefore expecting one or two arguments. With Babel, the above error message is replaced as follows:

```
=> (filter neg? 1 -1 0)
Wrong number of arguments in (filter neg? 1 -1 0): the
   function filter expects one or two arguments but was
   given four arguments.
In Clojure interactive session on line 1.
Call sequence:
[Clojure interactive session (repl)]
```

Moreover, the Babel spec for `filter` also reports incorrect types of arguments:

```
=> (filter neg? 3)
The second argument of (filter neg? 3) was expected to be a
   sequence but is a number 3 instead.
In Clojure interactive session on line 1.
Call sequence:
[Clojure interactive session (repl)]
```

Our project presently includes specs for many pre-existing Clojure functions and rewrites a majority of potential Clojure errors that cannot employ spec (such as divisions by zero, un-

matched delimiters, etc). With much of the internal work done at this stage of the project, our focus is now on the potential of third party tools to display modified error messages in an interactive viewer that will allow beginner programmers to explore them more dynamically and familiarize themselves with how Clojure itself handles error data, enabling them to view the data in a more organized way and easily explore related documentation.

## 3.2 Processing Error Messages

Babel constructs its error messages by intercepting exceptions directly from REPL. This is done by attaching an observer function to a running instance of a REPL server. The observer passes exceptions to babel processing function and replaces the message with the result. Before developing the interactive interface, the message was returned as a string. The processing for the interactive interface is described in Section 4.

Since different kinds of exceptions contain information in a different way, the first step in processing them is to determine what category they belong to. Errors can be categorized broadly into spec errors and non-spec errors

- Spec errors occur when the data does not conform to the specification for a function or a similar construct. These errors can be further divided into Babel specs, Clojure internal specs, and third-party specs. Babel specs were specifically designed by our group to provide as much information as possible to users, and in particular to novices. Clojure internal specs are provided for some specific Clojure constructs and provide useful information, although not necessarily phrased in a way suitable for beginners, so our software rewrites them to a more beginner-friendly format. Finally, third-party specs come with libraries imported into the code base. Since we don't know the semantics of these specs, we can only suggest what the intent is. Care should be taken not to second-guess, and thus obscure, the intent.
- Non-spec errors occur when errors happen outside of the clojure.spec system. They are typically either syntax error that happen before a function can even be called, or runtime errors that are not checked by spec, such as division by zero or any other errors that arise during program execution.

Due to the complexity of Clojure runtime system on the JVM, it may not be immediately obvious what type of error has occurred. For example, Clojure uses *lazy sequences*: many functions that return a sequence, such as `filter`, don't evaluate it immediately. The result is evaluated only when it's needed, and only as many elements as needed are evaluated. This may result in evaluating the sequence at the :print-eval-result phase. If an error occurs at that point, it may manifest itself as a nested error (an exception attached as a cause to another exception) since it occurs within a printing function.

In order to determine the type, the following properties of exceptions are used: *error types*, *levels of nesting*, and *phase*.

- Grouping by error types or subtypes can help find patterns in errors that make it easier to prescribe solutions to errors, based on similar error types.

- As we mentioned before, an exception may be nested within another exception if it is produced by a lazy sequence or in some other cases. Grouping by levels of nesting and the order in which errors occur provides more insight into the path the error took before it becomes visible, helping us understand the sequence of failures.
- Classifying by phase (such as :read-source, :execution, and :print-eval-result) offers additional context about when and where the error occurred.

Using the above in conjuction with each other allows us to find the elements of an exception that can be used to explain its cause to the user.

# 4 Adding Visual Interactive Environment

We explore using the third-party Morse viewer [9] as a supplementary way to interactively present error information in a better, more beginner-friendly format. This approach provides an interactive, visual layer of error message presentation on top of the existing REPL error message display. For instance, unlike default Clojure error messages, the Morse viewer allows us to use monospace font for code snippets, and highlight function names with color coding. Additionally, for spec errors, we can dynamically generate the link to the documentation of the function where the error occurred, provided it is a predefined Clojure function.

Morse is a tool designed for data exploration and visualization. While it was originally designed to help experienced developers navigate large datasets and projects, it provides tools and extensibility to be curated with a beginner audience in mind. Morse uses a system of customizable viewers to achieve its data visualization. Each view is individually designed to display a specific data structure based on given conditions.

## 4.1 Running a Babel Session with Morse

By default, the Clojure REPL does not expose the correct hooks to effectively manipulate the data from error messages. In order to get around this, initializing Babel within an existing REPL session, using a custom-made `(babel/init)` function to connect the processing hooks, creates a new custom sub-REPL (a new REPL session) above the existing session. Creating a sub-REPL allows us to configure the behavior attached to the following hooks in the new session:

- `:init` Defines the initialization behavior of the sub-repl. In Babel, this starts a new Morse session connected to the new REPL.
- `:eval` Defines the behavior whenever a command is executed within the REPL. In Babel, this taps the command into an atom (a mutable state variable), and evaluates the command within both Morse and the REPL.
- `:caught` Defines the behavior on exception. In Babel, this processes the error as described previously, and passes the following information about the error to Morse to be displayed 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 caused the error.

Once the sub-REPL is initialized, the user can then use the sub-REPL as usual, but with the addition of Babel error messages when applicable in the terminal, and a parallel pop-up Morse viewer that format the error message in an easier-to-parse way, providing interactive display for the user, for example giving them the ability to click on the documentation link.

In order to provide a message in the format suitable for More viewer display, Babel processing module returns the message as a sequence of key/value pairs, where each key indicates what part of error message it is. Fir example, the code part of the message will be labeled as `:code`, which allows displaying it in the monospace font common for code.

## 4.2 Setting up Morse Viewers

Morse viewers can be created to have custom behavior around certain data. Creating a new viewer in Morse requires the following:

- A predicate defining the constraints of the data that can be displayed in the viewer.
- The graphical format of the view to be displayed. This is built with the JavaFX library.

The Babel viewer takes a map of input data as input, and defines the predicate for the viewer by checking for a map with the necessary fields. For the formatting definition, Babel uses the built-in `WebView` object in order to format the error data with an HTML template, enabling us to leverage the large, robust tools of HTML and CSS to define the visuals of the error messages.

## 4.3 Exploring Design of Error Messages Interface

An example of the error presentation for Babel is shown in Figure 1, displaying the error message for the expression `(even? 1 2)`. Code segments are displayed as monospace and isolated. Additionally, function names within the body of the error message are color coded.

# 5 Current State and Future Work

Our project is currently focusing on the goal of utilizing Morse viewers for creating intuitive, robust interactive display of error messages. A lot of effort was put into figuring out the technical setup and data flow to accomplish this task. It also required changing the

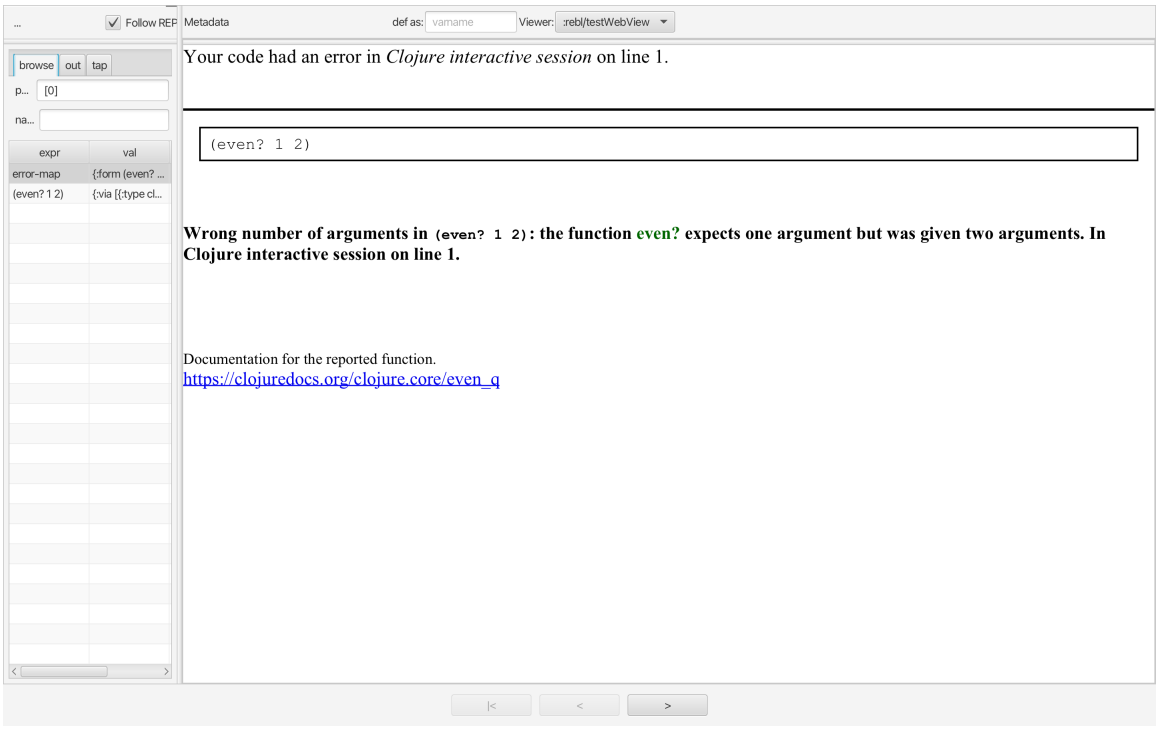


Figure 1: Morse visualization of the arity error on `even?` with too many arguments (right), with session evaluation history (left).

way an error message is passed to the display by adding tags (such as `:code`) for various elements of an error message.

We have accomplished creating a prototype for these interactions and the data flow. However, as the project stands right now, many features of the interactive display are still in active development:

- More work needs to be done to convert all of the exception processing into the tagged format.
- The ability to hover over specific terms used in error message for explanations of what they mean, as well as to navigate collapsible stack traces, would be useful for making the tool more interactive.
- Figuring out the user's workflow in alternating between correct code and errors is also a future work item.

In the future, we would like to conduct usability studies with the interactive tools offered by Morse, after we have developed the tool enough to cover more of Babel's capabilities. Allowing research participants to experiment with the tool for learning Clojure would help drive the direction of the project and improve upon the design choices. We would also like to explore IDE integration as a potential avenue of development for Morse, in order to make the tool more useful for learning to code as a whole.

# 6 Acknowledgments

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

# References

[1] Racket, the programming language. [https://racket-lang.org/, Online; accessed 3-22-2025].

[2] Stack overflow 2024 developer survey. [https://survey.stackoverflow.co/2024/, Online; accessed 3-22-2025].

[3] ABELSON, H., AND SUSSMAN, G. J. *Structure and Interpretation of Computer Programs*, 2nd ed. MIT Press, Cambridge, MA, USA, 1996.

[4] DONG, T., AND KHANDWALA, K. The impact of "cosmetic" changes on the usability of error messages. In *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2019), CHI EA '19, Association for Computing Machinery, p. 1–6.

[5] FELLEISEN, M., FINDLER, R. B., FLATT, M., AND KRISHNAMURTHI, S. The structure and interpretation of the computer science curriculum. *J. Funct. Program. 14*, 4 (July 2004), 365–378.

[6] HICKEY, R. clojure.spec - rationale and overview. [https://clojure.org/about/spec, Online; accessed 3-22-2025].

[7] HICKEY, R. The clojure programming language. In *Proceedings of the 2008 symposium on Dynamic languages* (New York, NY, USA, 2008), DLS '08, ACM, pp. 1:1–1:1.

[8] MARCEAU, G., FISLER, K., AND KRISHNAMURTHI, S. Mind your language: on novices' interactions with error messages. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (New York, NY, USA, 2011), Onward! 2011, Association for Computing Machinery, p. 3–18.

[9] TEAM, C. Introducing morse. [https://clojure.org/news/2023/04/28/introducing-morse/, Online; accessed 3-22-2025].

[10] WRENN, J., AND KRISHNAMURTHI, S. Error messages are classifiers: a process to design and evaluate error messages. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (New York, NY, USA, 2017), Onward! 2017, Association for Computing Machinery, p. 134–147.