Designing a Comparative Usability Study of Error Messages

Henry Fellows, Thomas Hagen, Sean Stockholm, and Elena Machkasova
Computer Science Discipline
University of Minnesota Morris
Morris, MN 56267
fello056@morris.umn.edu, hagen715@morris.umn.edu,
stock424@morris.umn.edu, elenam@morris.umn.edu

Abstract

Error messages are the only form of response that programmers get from malfunctioning programs. More experienced programmers often develop intuition about what error messages actually mean, but novices only have the content of the error message. Our research focuses on two functional programming languages in the Lisp family, their current or potential use in introductory CS classes, and specifically on the quality of their error messages for beginner CS students.

The languages we will be comparing are a subset of Racket called beginning student language, and Clojure. Beginning student language is a language designed for introductory students using the How to Design Programs 2 curriculum, with error messages that are designed for novices. Clojure is a Lisp built on top of the Java programming language which better supports concurrent and parallel programming and has been rapidly gaining popularity in industry. However, Clojure was not developed with beginner programmers in mind: its native error messages are often just Java error messages that dont make sense to programmers without Java background. Our previous work with Clojure has built an alternative error messages system that we think may be more useful to beginner CS students.

Our work is a part of the ClojurEd project which aims to use Clojure to teach an introductory CS course. One of the project goals is to provide introductory students with understandable error messages. The current stage of the project is to evaluate how well the new messages work for beginner programmers. We have designed a usability study comparing the proposed and standard Clojure error messages to each other and to Racket error messages. In this paper we present the details of the study and the approaches to developing code samples that allow us to compare error messages systems. We also present and discuss preliminary results of the usability study.

1 Introduction

Error messages are the primary form of response that programmers get from malfunctioning programs. Experienced programmers have the background to understand what error messages actually mean, but novices only have the content of the error message. While interactions with errors in a programming language are an essential part of a programmer's experience, quite often error handling in a language is not very well thought through: error messages use inconsistent terminology, are ambiguous or misleading, and do not provide enough detail. Beginner programmers find error messages unhelpful. Anecdotal evidence (informal questioning of about 10 CS students and almuni by the authors) indicates that novice programmers often do not even realize that an error message may contain useful information, except perhaps the line number; some ignore these messages entirely. This observation is consistent with the general programming languages community undervaluing of error messages quality. There has been very little formal study of usability of error messages for beginners, and very little methodology developed for such studies. The notable exception is the development and study of beginner-friendly error messages in the Racket programming language [5, 6].

Our work focuses on error messages for the Clojure programming language. Clojure is a functional language in the Lisp family released in 2007. Clojure is compiled to Java bytecode, employing the Java Virtual Machine (JVM) as its interpreter. It features Java interop: the ability to use Java code directly in Clojure code. It quickly gained popularity, is widely used in industry and in open-source projects, and features a large community of developers with three major annual conferences (two in the U.S. and one in Europe), as well as numerous smaller gatherings and meetups.

As a Lisp, Clojure has simple syntax and dynamic type system, and therefore may be a promising option in a functional-first approach to teaching programming, e.g. [7]. However, due to its implementation over Java, Clojure is notorious for its cryptic and unintuitive error messages. Clojure errors are just Java exceptions, and therefore are phrased in terms of Java types (e.g. `java.lang.Number`) and Java operations, such as typecasting. They also come with Java stack trace which can be particularly verbose in Clojure since functional languages tend to have a large number of function calls on the stack. Clojure error messages are often confusing even to experienced programmers, and are a barrier for beginners.

In the last few years there has been a large community effort to address this issue. Several Clojure IDEs come with their own error processing system that filters our Clojure internal functions from the stack trace. There has been work done in Clojure community on improving error messages for macros [4]. However, these improvements address the needs of more experienced programmers by continuing using Java types and more advanced terminology, and leave out those for whom Clojure would serve as the first introduction to programming. By contrast, our work on error messages aims at addressing beginners' needs.

1.1 Our project and the study research goals

This work is a part of a larger project on making Clojure more accessible to beginners. As a part of this project, we are developing an alternative system of error messages that do not use Java datatypes or other terminology that beginner programmers would not be familiar with. For instance, an erroneous fragment `(+ [1 2]))` is attempting to use a + function on a vector of numbers `[1 2]` instead of the numbers themselves. This results in the following error in Clojure:

```
java.lang.ClassCastException: Cannot cast
clojure.lang.PersistentVector to java.lang.Number
```

Neither the datatypes nor the term "cast" are familiar to beginners.

Our error message for the same fragment is

```
In function +, the first argument [1 2] must be a number
but is a vector.
```

Some of our alternative error messages are described in [3]

While these messages seem to be an improvement, we would like to formally examine two hypotheses about our alternative system of Clojure error messages:

1. It is easier to understand than Clojure native error messages.
2. It is at least as helpful as the system of error messages in the Racket programming language.

To test these hypotheses, we have developed a comparative usability study that compares our system of error messages to the native Clojure error messages and to those in the Racket programming language. We use a series of erroneous code fragments that test subjects need to correct, and measure how many fragments they have successfully fixed and how many steps it took them to fix the issue or make edits that make progress towards the solution. Study subjects are undergraduate students with Racket background who have none, or very little, Clojure background. Detailed description of the study is given in Section 3.2.

A statistically significant higher number of fixed code fragments and a lower number of attempts for our error messages system, compared to the native Clojure system, would indicate that our error messages are easier for students to understand. Additionally we are measuring the difference between either one of the Clojure error messages systems and the Racket messages. Racket error messages were specifically developed to be helpful for beginners. Thus, a comparison to Racket messages would indicate how our error messages compare to the expectations of error messages for beginner-friendly functional languages.

Currently the study design is mostly completed, but the study itself has not been conducted yet. The paper describes the study design goals, as well as its setup and questions.

The rest of the paper is structured as follows: we give an overview of Lisps in general and Racket and Clojure in particular in Section 2, explain the setup of the study in Section 3,

focus on questions selection in Section 4, give examples of the questions in Section 5, and present conclusions and discuss future work in Section 6.

2 Languages

The two languages we are working with (Racket and Clojure) belong to the Lisp family of languages.

2.1 Overview of Lisp

Lisp is a family of functional programming languages that dates from 1958, developed by John McCarthy based on Alonzo Church's lambda calculus. Lisp was the first language in which data and program code were internally stored using the same data structure which opened possibilities for manipulating not just data, but also program code, within a program. Lisp has a distinctive fully parenthesized prefix notation, where all code and data are written as expressions in the form:

```
(<function-name> <arg 1> <arg 2> ... <arg N>)
```

Almost all operations are functions. Addition, for instance, is a function that can be applied to any number of arguments.

```
(+ 2 4 6 8)
-> 20
```

Here `->` indicates the result of computations in the Lisp interpreter.

Lists in Lisp are commonly created by using the list function

```
(list 2 4 6 8)
-> '(2 4 6 8)
```

The return value `'(2 4 6 8)` uses the single quote `'` to indicate that the expression should not be evaluated (doing so would attempt to apply the number `2` as a function), but instead it should be treated as data storage.

```
(list 2 4 6 8)
-> '(2 4 6 8)
```

The use of Lisp in teaching computer science starts with the Structure and Interpretation of Computer Programs, a 1985 textbook that aimed to identify general patterns instead of learning the details of a specific programming language. Structure and Interpretation of Computer Programs used Scheme, another variant of Lisp as the language of instruction. The influence of this textbook continues in the form of How to Design Programs, which is the textbook that uses Racket. Felleisen et al [2] have made an excellent case for using a Lisp as a programming language in an introductory CS class as the first language new

programmers will use. Lisp offers a simple syntax and introduces students to modularity, abstraction, and data-driven program design while developing good programming practices by being explicit about program design principles. Teaching these concepts in introductory courses creates a foundation that later classes can build upon in while teaching popular imperative and object-oriented languages. [1].

2.2 Differences between Racket and Clojure

Racket and Clojure, while both Lisps, satisfy different design goals. Racket features a set of language levels with limited subsets of the full language that allow instructors to slowly introduce features to students.Clojure focuses on concurrency and integration with the Java virtual machine. The differing purposes of these language have created differences in the syntax and idioms of Racket and Clojure. The differences begin at definition of variables; Clojure defines functions with the following syntax

```
(defn function-name [<arg 1> ... <arg N>] (expression))
e.g
(defn plus-four [x] (+ x 4))
```

Racket uses this syntax:

```
(define (function-name <arg 1> ... <arg N>) (expression))
e.g
(define (plus-four x) (+ x 4))
```

Racket and Clojure both have associative datatypes.We used Clojure hashmaps in the study because they are the most common associative datatype in Clojure. Hashmaps in Clojure are a collection of key-value pairs:

```
{key value, key value, key value}
```

Hashmaps commonly use keywords. Keywords are simple names that have a colon in front. Keywords are often used as keys within hashmaps. When used as a function on a hashmap, keywords return the value associated with the keyword. An example of a hashmap using keywords follows:

```
{:a 1, :b 2, :c 3}

(:a {:a 1, :b 2, :c 3})
-> 1
```

The keywords in the above hashmap are: `:a`, `:b`, and `:c`.

In the above example, `:a` is bound to `1`,`:b` is bound to `2`, and `:c` is bound to `3`.

Racket uses structures as its associative datatype, which has the following syntax:

```
(struct position (x y))
```

```
(position 75 42)
-> #<position>
```

Structs automatically generate an accessor function, such as `position-x` for each field in the structure. Racket's `struct` defines a type of structure and the name of it's fields. The name of the struct is then used as a constructor function. Unlike hashmaps, structs cannot be extended at runtime; elements can be removed, added, or replaced in hashmaps, but not in structs.

```
(position-x (position 75 42))
-> 75
```

Both Racket and Clojure have `let`, which provides local bindings within the scope of let. Clojure's `let` has the following syntax:

```
(let [x 1 y 2]
  (+ x y))
-> 3
```

Racket's syntax is similar, but each pair of symbols and expressions is placed in brackets.

```
(let ([x 1] [y 2])
  (+ x y))
```

Another important syntactic difference is that Racket uses the keyword `Lambda` for defining anonymous functions, whereas Clojure uses `fn`.

3 Usability Study

3.1 Study participants

The usability study that we have developed is structured around "parallel questions": erroneous fragments that are the same in Racket and Clojure (except the syntactic differences), but result in different error messages. The participants for the study are volunteer students from the U of M, Morris who have taken the introductory computer science course in Racket and have no, or very little, experience with Clojure. These students have reasonable knowledge of Racket and functional approaches to problem solving while still being beginners in terms of programming practice and experience. We expect to have about 20 participants. Participants will be recruited via the CS department mailing list, as well as via email and short presentations in freshmen and sophomore level CS classes. In order to attract a larger and more diverse student population for the study, students will be compensated for their time, thanks to a gift from Cognitect, Inc. The study is exempt from full IRB review.

3.2 Experimental Setup

The study is split into two main parts: Racket review and testing, and a brief overview of Clojure and testing. The goal of each section is to introduce the two languages' basic functionality and then test the participants ability to correct mistakes in that language. Testing is followed by a brief "interview" at the end.

Below is the outline of a participant's experience:

- Participant will be assigned an instructor
- Participant will be given a review of Racket
- Participant will be given a series of program fragments in Racket to correct
- Participant will be presented with a brief overview of Clojure
- Participant will be given a series of program fragments in Clojure to correct. In this part each participant will randomly receive only our error messages or only Clojure's default error messages
- The errors participants create and their current code will be captured via screencap at a regular intervals
- Each instructor will document their participants actions and errors in general
- The participants will be asked a series of questions about the tests at the end

3.2.1 Languages overview

The overall purpose of each instructional session is to ensure that the participants have enough fundamental knowledge of the language presented to them to have a basic understanding of the problems. In addition, the session should also ensure a relatively similar level of familiarity with both Clojure and Racket among all participants. Each participant in the study will go through the process of reviewing and testing individually in order to minimize outside influence from other participants or distractions. Because of this, the participant's experiences must be parallel with each other both in terms of the environment and the materials provided such as not to create bias.

The Racket review section will be a reintroduction to participants, as they should already have a base knowledge of Racket from previous courses taken. The participants should have little or no knowledge of Clojure when coming into the study, so the Clojure lesson will focus more on drawing similarities from Racket and basic function explanations. The core topics to cover for each language are syntax (such as the prefix notation), higher order functions, function definitions, and recursion. This part will consist of a brief writeup with examples. The Clojure writeup will assume familiarity with Racket and will highlight the differences.

3.2.2 Testing

After the overview the participants will be assigned to a computer where their actions will be monitored by a reviewer. From here, they will be directed to connect to a website we have constructed that will present them with a series of problems to solve. Participants will not be allowed to consult outside sources, the reviewer, or other participants during this period of the study except for the language's public API for Racket and Clojuredocs for Clojure. The participants will also have access to Racket's built-in IDE for Racket questions and to a project in the LightTable code editor for the Clojure questions.

The website they connect to pulls its questions from a set of parallel questions between Clojure and Racket we have constructed, so that for every question in Racket there is an equivalent question in Clojure and vice-versa. Each question pair has a difficulty rating associated with it, with 1 being the easiest questions and 5 being the hardest questions. When a new participant logs into the website, it constructs both the Racket and Clojure tests to present by randomly selecting question-pairs from the pool. The program must choose set amounts of questions from each difficulty, such as three level-1 questions, one level-2 question, two level-3 questions, etc., every time it creates tests.

As an example of this, assume a student sits down and logs into the website to take a test. The website goes to construct the test under the restriction that it should try to assign equal amounts of question from every level between each test, and it should use every question. It checks the pool of level one questions and sees there are seven possible level one question-pairs for the program to use. We can think about the pool like this,

LEVEL 1: (C1-1,R1-1), (C2-1,R2-1), (C3-1,R3-1),...,(C7-1,R7-1)

where a question pair is C1-1 (Clojure question 1, level 1) and R1-1 (Racket question 1, level 1), which are essentially the same question in different languages.

The website then randomly selects three level one question-pairs from the seven, removes them from the pool, and adds the Racket version of those questions to the set of Racket questions for the test. The website constructs the rest of the Racket test in a similar fashion for each difficulty level by randomly selecting half of the question-pairs, adding the Racket version to the Racket test, and removing the pair from the pool. Even though we are only using the Racket version of the question in the Racket test, we still remove the whole pair from the pool so that we don't end up with question R1-1 in their Racket test and C1-1 in their Clojure test, giving them the same questions twice. After the Racket test is made, the website takes the rest of the questions in the pool and constructs the Clojure test from them. In the end our tests for this participant may end up looking something like this;

Racket Test: R1-1, R6-1, R3-1

Clojure Test: C2-1, C5-1, C7-1, C4-1

The participants will be allocated a time limit of 20 minutes to answer the questions in each language. They are allowed to try as many times as needed on each questions and they are allowed to skip past questions with the possibility of coming back to them later on.

Before taking their Clojure test each participant will be assigned to receive either Clojure's default error messages or our custom Clojure error messages. While testing, the computer will take a screenshot periodically, and the reviewer will take notes on what the participant does while solving the problems. The purpose of doing both is so that later on we know what to look for in screenshots. Screenshots in which there are no changes will be discarded. Each session will result in a series of screenshots and a short "script" of the participant's actions with timestamps in minutes and seconds since the beginning of the test, e.g. "At 1:12 ran the program, got a message `Cannot convert 2 to a function`, at 1:45 switched the order of the function parameters", etc. After the finishing all of the questions, the participant will be asked a series of questions related to the tests such as, "What part of the tasks were easy for you? What parts were hard?".

3.2.3 Data evaluation

After the question period of the study, we will now have a collection of screen captures from each participant detailing the process solving the problems, notes on their problem solving process, and participant opinion on the challenges of the study. From these we can begin to gauge the significance of the error messages in their problem solving processes. Numerical data (the number and the level of the solved problems, total time until correct solutions, the number of edits towards the right solution vs those leading away) will be processed statistically.

The questions at the end would allow us to gauge how the messages are perceived, and whether this correlates with their actual usefulness. A positive correlation would be useful to know for future research, allowing us to rely on participants' feedback. Lack of such correlation would mean that participants' perception is unreliable and should not be used for future evaluation.

4 Principles of Question Selection

Selecting questions for the study was a challenging process since every question must make sense in both Racket and Clojure and not require an involved introduction. Below we detail our approach to question selection.

4.1 Selecting Meaningful Accessible Questions

When putting together questions for the study, we followed the following principles:

- The code examples should be simple enough to understand the programmer's intent with just a few test examples.
- The code examples must have mistakes that a beginner programmer would make. Examples include switched arguments of a function or a mistyped identifier.

- There must be a simple fix for an error. This is challenging since some errors, such as s type mismatch, can be caused by a variety of issues, and a beginner programmer can easily start making changes that are more complicated than they need to be, in an attempt to match the types.
- Code examples and tests must use the same simple set of features in Racket and in Clojure. For instance, Racket uses a function `check-expect` for testing. Clojure also has a similar library (expectations), but it is more convoluted. Thus in order to equalize the experience in the two languages we chose to use only = and `equals?` function for testing since they are the same in the two languages.

4.2 Question Parallelism

One of the overarching themes of this study was the difficulty in creating questions that had similar meaning, errors, and syntax in both Clojure and Racket. Often, our questions are somewhat less than idiomatic because of the constraints imposed by the short time commitment we intended for the study. For instance, we only use lists in our questions, where idiomatic Clojure prefers vectors, which combine the best features of arrays and lists. Reducing the amount of review and potential sources of confusion was paramount; For instance, Clojure doesn't support tail recursion properly because of the limitations of the Java virtual machine. In order to create tail-recursive structures, idiomatic Clojure uses a construct called loop recur to bypass this limitation. Loop recur has complex syntax and semantics, and so our questions avoid the use of Loop recur, despite the fact that it is idiomatic. The gap between idiomatic code and testable code is wide, but avoiding complex features is a good start.

5 Examples

In this section we provide two examples of questions in the study.

This first example is categorized as a medium level of difficulty. It asks the participant to make the function work so that all the test cases below evaluate to true. The function is given in Racket as

```
(define (filter-even elements)
  (foldl (lambda (x y) (if (even? y) (cons x y) y)) '() elements))
```

and in Clojure as

```
(defn filter-even [elements]
  (reduce (fn [x y] (if (even? x) (cons y x) y))  '() elements))
```

In this case, Racket and Clojure have identical test cases:

```
(= '(2 4 6 8) (filter-even '(1 2 3 4 5 6 7 8 9)))
```

```
(= '() (filter-even '(1 3 5 7 9)))
```

It is at this difficulty because of the use of a higher order function. We would like to see if the error message can help the subject to recognize that we should be testing the parity of the element we are adding to the list, and not the parity of the list itself. The error in this example is can be fixed by replacing `(even? y)` with `(even? x)` in Racket and `(even? x)` with `(even? y)` in Clojure. Note that Racket `foldl` and Clojure `reduce` take their arguments in the opposite order. However, the test subjects will not see these two examples together (they will get either a Racket version or a Clojure version), and thus would not be confused because of inconsistency.

The unmodified Clojure error in this case is:

```
java.lang.IllegalArgumentException: Argument must be an integer:
clojure.lang.PersistentList$EmptyList@1
               core.clj:1355 clojure.core/even?
```

Our error is:

```
Error: In function even?, the first argument () must be an integer
number but is a list.
```

Racket error:

```
even?: expects integer, given '()
```

We have chosen careful wording to communicate that the `even?` requires an integer as its first argument, but was given a list. Our error message is clearly an improvement on standard Clojure, since we have told the user the type expected. It also gives more detailed information than the Racket example, since it tells the user the type of the wrong argument it received.

We have categorized the following second example is an easy problem:

Racket code:

```
(define (my-length elements)
   (cond
   [(empty? elements) 0]
   [else (+ 1 (my-length (first elements)))]))
```

Clojure code:

```
(defn my-length [elements]
  (if (empty? elements) 0 (+ 1 (my-length (first elements)))))
```

The test cases are:

```
(= (my-length '(5 4 3 2 1)) 5)
(= (my-length '()) 0)
(= (my-length '(1 3 5 7 9 11)) 6)
```

The error is caused because first requires a sequence as its first argument. The error in this example can be fixed by replacing the function `first` with `rest`. The Clojure code in this example is done in a non-ideomatic way so that it is more comparable to the Racket code. We hope our error message will help the subject to realize that they should be dealing with a sequence, not an element of a sequence.

The standard Clojure error:

```
IllegalArgumentException Don't know how to create ISeq from:
java.lang.Long  clojure.lang.RT.seqFrom (RT.java:528)
```

Our error:

```
Error: In function first, the first argument 5 must be
a sequence but is a number.
```

Racket error:

```
first: expects a non-empty list; given: 1
```

An introductory level student would not be able to parse much, if any useful information out of the original Clojure error in this case. Our error clearly communicates that the first argument of first must be a sequence but is a number. Racket also communicates that information, but with fewer words.

6 Conclusion and Future Work

At this stage of our work we have developed the study setup, the principles for question selection, and developed questions for the study, as well as interview questions for participants. We also have narrowed down the set of features of the two languages that our study uses. As far as we know, this is the only comparative study of error messages usability between two different languages, and we have encountered (and overcome) challenges in making the questions simple, clear, and parallel in the two languages. We are still working on putting together the overview of the two languages, now that we have narrowed down what these overviews include.

The major task that remains to be done is conducting the study itself and processing its results. We hope to have it completed by the end of April 2016.

The completion of the study will point out strengths and weaknesses of our error message system, which would direct our future research.

7 Acknowledgments

Thanks to UMN UROP, UMM MAP, and UMM LSAMP for providing support for students research, and to Cognitect, Inc. for providing funding to compensate study participants.

References

[1] BIENIUSA, A., DEGEN, M., HEIDEGGER, P., THIEMANN, P., WEHR, S., GASBICHLER, M., SPERBER, M., CRESTANI, M., KLAEREN, H., AND KNAUEL, E. Htdp and dmda in the battlefield: A case study in first-year programming instruction. In *Proceedings of the 2008 International Workshop on Functional and Declarative Programming in Education* (New York, NY, USA, 2008), FDPE '08, ACM, pp. 1–12.

[2] 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.

[3] FELLOWS, H., LEMMON, A., MAGNUSON, M., SAX, E., SCHLIEP, P., AND MACHKASOVA, E. Developing beginner-friendly user interactions for the clojure programming language. MICS'15.

[4] FLEMING, C. Improving Clojure's error messages with grammars, a presentation at Clojure/conj, November 2015.

[5] MARCEAU, G., FISLER, K., AND KRISHNAMURTHI, S. Measuring the effectiveness of error messages designed for novice programmers. In *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education* (New York, NY, USA, 2011), SIGCSE '11, ACM, pp. 499–504.

[6] 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, ACM, pp. 3–18.

[7] THOMPSON, S., AND HILL, S. Functional programming through the curriculum. In *Functional Programming Languages in Education* (December 1995), P. H. Hartel and R. Plasmeijer, Eds., no. 1022 in Lecture Notes in Computer Science, Springer-Verlag, pp. 182–196.