# FSM Error Messages

Marco T. Morazán
Seton Hall University
morazanm@shu.edu

Josephine A. Des Rosiers
Seton Hall University
desrosjo@shu.edu

Computer Science students, in general, find Automata Theory difficult and mostly unrelated to their area of study. To mitigate these perceptions, FSM, a library to program state machines and grammars, was developed to bring programming to the Automata Theory classroom. The results of the library's maiden voyage at Seton Hall University had a positive impact on students, but the students found the library difficult to use due to the error messages generated. These messages were generated by the host language meaning that students needed to be familiar with the library's implementation to make sense of them. This article presents the design of and results obtained from using an error-messaging system tailor-made for FSM. The effectiveness of the library was measured by both a control group study and a survey. The results strongly suggest that the error-messaging system has had a positive impact on students' attitude towards automata theory, towards programming in FSM, and towards FSM error messages. The consequence has been a marked improvement on students' ability to implement algorithms developed as part of constructive proofs by making the debugging of FSM programs easier.

## 1 Introduction

Programming-oriented students usually find automata theory, grammars, constructive proofs, computability, and decidability challenging and in many cases overwhelming. The existence of this perception is not completely absurd since students are asked to design and prove correct machines/grammars/algorithms without being able to experiment nor get immediate feedback, as they do in any programming course. Both of these are essential in a learning context and need to be timely (the sooner the better) [11]. In fact, designing programs without being able to experiment and to get immediate feedback in the form of *understandable* error messages goes against the grain of what students learn in programming courses. Given that formal automata textbooks (e.g., [3, 6, 13]) rarely offer any software infrastructure for students to experiment with their ideas and designs, it comes as no surprise that students dislike the material. More importantly, however, this has an impact on the quality of the constructive proofs they develop and, therefore, the grade they receive. Students may have a general idea for an algorithm, but are fuzzy about the details. Having them code the machines/grammars/algorithms they develop (as part of a constructive proof) forces them to think about the details and improves the quality of their proofs resulting in better grades and less frustration.

Given that it is reasonable for Computer Science students to be able to experiment with their designs, the FSM (**F**unctional **S**tate **M**achines) library was developed [8]. This library (implemented in Racket) allows students to implement state-machines (e.g., finite-state automata, pushdown automata, and Turing machines) and grammars (e.g., regular, context-free, and context-sensitive). In addition, the library implements many common constructors (e.g., union, concatenation, and complement) derived from constructive proofs developed in class which students can use as part of their designs. The library also provides useful functions, like renaming the states of a machine and renaming the nonterminals of a grammar, to allow students to focus on the algorithms they develop as part of their constructive proofs

P. Achten, H. Miller (Eds.):
Trends in Functional Programming in Education 2018 (TFPIE'18)
EPTCS 295, 2019, pp. 1–16, doi:10.4204/EPTCS.295.1

instead of focusing on the auxiliary functions they may need. Finally, FSM provides random testing facilities for both machines and grammars which students (and instructors) can use to validate (not verify) a machine or grammar. The expectation is three-fold. First, we expect CS students to develop a more favorable attitude towards the material as they are able to implement and test their algorithms. To the best knowledge of the authors, FSM is the only library of its kind that provides students with an appropriate level of abstraction to achieve this. That is, students do not need to, for example, wrestle with how to implement nondeterminism. Second, given that students can debug their algorithms before submitting them for grading the expectation is that more solutions to problems will be correct. Third, grading is simplified for the instructor by being able to test the, sometimes convoluted, solutions students develop.

The maiden voyage of FSM at Seton Hall University (SHU) took place in the Spring semester of 2016 and was taught by the first author. The results, at best, did not fully meet the expectations. Although students seemed to develop a more favorable attitude towards Automata Theory, there was still a significant number of students that simply did not relate to the material. Grades were also, generally speaking, higher thanks to the library, but mostly for students that made an effort to see the instructor outside of class or to ask questions during class. The overwhelming number of questions were about how to use FSM. Finally, grading was not significantly simplified. In many instances, it was very difficult to explain why the constructors for state machines and for grammars submitted by students were buggy.

The lukewarm results for all three expectations were rooted in the lack of a proper error-messaging system for FSM. In general, the errors that students (and the instructor) received came from the Racket error-messaging system and were rarely useful in diagnosing the error at the FSM level. For example, consider applying a machine M to a word w. If M is in state A and the next element of w to be processed is b, FSM filters M's rules to find the ones that apply to such a configuration. The rules are then applied to create a list of next possible configurations (recall that M may be nondeterministic). A student, however, has mistyped the state A as a in the rules. Thus, the filtered list of rules is empty. This leads to a Racket error stating that car cannot be applied to the empty list when FSM attempts to create the list of next possible configurations. To understand such an error, the programmer has to be intimately familiar with the implementation of FSM, which is an unreasonable expectation for students in the course (just like being familiar with the implementation of Racket is also unreasonable). A more useful error ought to be produced by the library when the constructor for the machine was called with rules that have typos.

This article presents the design of an error-messaging system for FSM and the impact it has had on students. The focus is on generating error messages that are meaningful to students when they improperly use a constructor. Section 2 reviews related work on designing error messages. Section 3 provides a brief overview of FSM. Section 4 presents examples of code developed by students and the Racket error messages they originally received. Section 5 presents the design of FSM error messages. Section 6 presents feedback received from the students. This section is divided in two subsections. Section 6.1 presents the results obtained from a control-group study performed by asking students to debug the solutions to three problems. The control group worked with a version of FSM that did not have a specialized error-messaging system and the focus group worked with one that did. Section 6.2 presents results obtained from surveying the students. Finally, Section 7 briefly presents some conclusions and directions for future work.

## 2 Related Work

There are two primary sources in the literature that influenced the design of the FSM error-messaging system. The first is the work done by developers of programming languages for students. The second is the work done by the human-computer interaction (HCI) community on compiler error messages. Both communities have developed principles to guide implementors of error-messaging systems. These communities coincide in stating that experienced programmers can often see mistakes that students do not and in stating that error messages need to be understandable [1, 2, 9, 12, 14]. These two conclusions come as no surprise, but how to make error messages understandable, especially to students, is yet unresolved.

A fundamental principle is that how error messages are phrased is important [7, 14]. For students, this means that error messages must not be filled with technical jargon that is unfamiliar to them. In fact, error messages must be recognized as the primary feedback mechanism for students [4, 5, 9]. This is evident, because when solving homework problems, students spend more time wrestling with error messages than they do asking an instructor what an error message means. To assist novice programmers it is important for error messages to be expressed using the same vocabulary that is employed in class by the instructor [4]. In this manner, error messages may become meaningful to students by eliminating unfamiliar technical jargon. The work presented in this article is very much inspired by this principle. The vocabulary used by the instructor in class is the same vocabulary that students see in an error message. Furthermore, the instructor on occasion purposely presents buggy code to the students in order to demonstrate what the error messages mean and to explicitly show students that the vocabulary in the error messages is the vocabulary used in class. The instructor makes a conscious effort to use terms that appear in error messages (e.g., states, alphabets, and rules) instead of using technical jargon that explains why in the FSM implementation incorrect inputs to a constructor lead to errors when applying machines and grammars. The set of terms used are unified over the textbook, the error messages, and the lectures.

Error messages must also be carefully designed to prevent students from taking random actions to correct a bug [14]. To this end, it is important for error messages to provide guidance as to why the problem exists so that the student/programmer can make progress in resolving it [4, 5, 14]. Poor error messages (especially those that prescribe solutions) lead to actions that do not bring programmers, especially novices, closer to a solution. For example, FSM can be used to construct a deterministic finite-state machine where the set of states is '(X Y Z) and the alphabet is '(a b). A programmer may then mistakenly define a transition rule to move from state X to state Z on a as '(x a Z). An error message generated when the machine is applied to some input stating that the list of rules is empty is a poor message. It leads students to edit the design of their machines. A more useful error message is to have the constructor for deterministic finite-state machines throw an error stating that: *in the rule '(x a Z), x is not a valid state*. In fact an error message should not propose a solution [4]. For the example above, there are at least three possible corrective actions: add x to the set of states, change x to X in the rule, or change x for some other state in the rule. There is no way for the error-messaging system to know which corrective action should be taken. This principle is another of the guiding pillars of the FSM error-messaging system.

There is some debate on whether longer or shorter error messages are better. Some argue that longer error messages are not better [10]. Intuitively, this can be explained by students not willing to read error messages that they do not understand (i.e., the used vocabulary is foreign to them). If this is the case, then it is difficult to discern if the length of a message actually matters. On the other hand, short error messages may be too brief and not provide enough information to fix a bug. The problem is compounded when you need dependent types to describe the inputs to a function. For example, the set of rules given as input to the constructor for a deterministic finite-state machine depends on the given set of states and the given alphabet. If the set of rules is invalid, should an error message be short and simply state that the set of rules is invalid or should the error message be longer and state which and why rules are invalid? In FSM, we have chosen to have longer error messages. This decision is based solely on

```
(define sol1-dfa (make-dfa '(q0 q1 q2 ds) ;; the set of states
                           '(a b)         ;; the input alpahbet
                           'q0            ;; the starting state
                           '(q1)          ;; list of final states
                           '((q0 a q1)    ;; list of transition rules
                             (q0 b ds)
                             (q1 a q1)
                             (q1 b q2)
                             (q2 a q1)
                             (q2 b q2)
                             (ds a ds)
                             (ds b ds))))

(check-expect (sm-apply sol1-dfa '(a)) 'accept)
(check-expect (sm-apply sol1-dfa '(a b b b a)) 'accept)
(check-expect (sm-apply sol1-dfa '(a b a a b b)) 'reject)
```

Figure 1: FSM implementation of a deterministic finite-state machine to recognize a(a $\cup$ b)$^*$a.

informal discussions with students that state that if error messages were actually useful, then they would be willing to read them.

## 3 Overview of FSM

The FSM library presents the user with a generic interface to construct and manipulate state machines and grammars. Constructors are divided into two categories: primitive constructors and transformers. Primitive constructors build a state machine or a grammar from a formal description provided by the programmer. Transformers build a state machine or a grammar from existing machines or grammars exploiting algorithms obtained from constructive proofs. Observers are divided into three categories: accessors, applicators, and testers. Accessors return a specified component used to build a grammar or a state machine. Applicators apply a given machine or grammar to a word. Testers allow for machines and grammars to be tested with words provided by the programmer or with randomly generated words by the software. The latter two provide students with immediate feedback on the validity (not the verification) of their designs and implementations. In this section, we restrict ourselves to a couple of examples to give the reader a flavor of how FSM primitive constructors are used. For a full description of FSM, the reader is referred to a previously published article [8].

Let $\Sigma = \{a,b\}$ be an input alphabet. Consider the problem of recognizing the regular language:

$L = \{w | w \in \Sigma^* \wedge w \textit{ starts and ends with an } a\}$.

In FSM, the finite-state machine is constructed as displayed in Figure 1. We can describe each state with an invariant. Informally, q0 is the starting state representing that nothing has been read. The state q1 represents that the input read so far starts and ends with an a. The state q2 represents that the input read so far starts with an a and does not end with a. Finally, the dead state, ds, represents that the input does not start with an a. Based on these invariants, it is clear that the only final (or accepting) state can be q1. Each transition specifies a from-state, the input read on the tape, and a destination-state. For example, '(q0 a q1) says that from state q0 upon reading an a the machine moves to state q1.

The programmer can test the machines with, for example, 5 randomly generated inputs as follows:

```
> (sm-test sol1-dfa 5)
'(((b a a a a b b) reject)
  ((b a b b b a) reject)
  ((a a b a a a b a) accept)
  ((a b a a a b) reject)
  ((a a) accept))
>
```

As a second example, consider implementing a context-free grammar for the set of palindromes over the same Σ above. In FSM, the context-free grammar is constructed as follows:

```
(define palindrome (make-cfg '(S)    ;; the set of nonterminal symbols
                             '(a b)  ;; the alphabet
                             `((S ,ARROW ,EMP) ;; the set of rules
                               (S ,ARROW a)
                               (S ,ARROW b)
                               (S ,ARROW aSa)
                               (S ,ARROW bSb))
                             'S))   ;; the starting nonterminal
```

The constants ARROW and EMP are defined by FSM to denote $\rightarrow$ and, the empty string, $\varepsilon$, respectively.The programmer can now test the grammar with 3 random words as follows:

```
> (grammar-test palindrome 3)
'(((b b a a a) "(b b a a a) is not in L(G)")
   ((b a a b) (S -> bSb -> baSab -> baab))
   ((b b b) (S -> bSb -> bbb)))
>
```

As the reader can see, if a word is in the language the tester provides its derivation. Otherwise, the tester explicitly states the word is not in the language of the grammar.

## 4 Student Programming Examples with Racket Errors

This section presents and discusses the error messages obtained with three sample student programs that contain bugs. The error messages in this section are those obtained before the FSM error-messaging system was developed. You do not need to be familiar with Racket as the error messages are explained with a mostly jargon-free description.

Figure 2 displays the implementation of a Turing machine that decides the language of all strings in $(\texttt{a, b})^*$ that have two consecutive b. The student has correctly designed the transition function that moves from the starting state, S, on a b to a state, A, meaning that a b has been read. From A on a b the machine moves to a state, B, meaning that two consecutive b's have been read. From A on an a the machine moves to S meaning that no b's for bb have been read. Both states, S and A, on a blank correctly move to state N representing the reject state as bb is not contained in the input word. From state B, the machine reads the rest of the word and upon reading a blank moves to the accepting state, Y, and halts. It is unimportant that reading the rest of the input is unnecessary and a more "efficient" Turing machine would directly move from A on a b to Y and halt (eliminating the need for B). The student has even included a pair of tests to demonstrate their understanding of how the machine ought to behave.

```
(define has-bb (make-tm '(S A B Y N)          ;; the set of states
                        'S                    ;; the starting state
                        '(a b)                ;; the alphabet
                        '(Y N)                ;; the halting states
                        `(((S b) (A ,RIGHT)) ;; the transition rules
                          ((S a) (S ,RIGHT))
                          ((S ,BLANK) (N ,BLANK))
                          ((A b) (B ,RIGHT))
                          ((A a) (S ,RIGHT))
                          ((A ,BLANK) (N ,BLANK))
                          ((B a) (B ,RIGHT))
                          ((B b) (B ,RIGHT))
                          ((B ,BLANK) (Y ,BLANK)))
                        'Y))                  ;; the accepting state

(check-expect (sm-apply has-bb '(a b a a b a b b a a)) 'accept)
(check-expect (sm-apply has-bb '(a b a a b a a a)) 'reject)
```

Figure 2: Buggy Turing machine that decides the language L = {w | w has two consecutive b's}
.

The problem with this solution is that 4 of the arguments to the constructor, make-tm, are in the incorrect position. The correct order for the arguments is: the list of states, the alphabet, the transition rules, the starting state, the list of halting states, and the accepting state. When the tests are executed, the following error message is obtained:

```
caar: contract violation
expected: (cons/c pair? any/c)
given: 'a
```

Clearly, this error is not useful in determining that the arguments to the constructor are incorrect unless you are familiar with the implementation of FSM. FSM is reporting (via the Racket error-messaging system) an error that occurs while attempting to parse the argument given for the list of transition rules. There is no way for a programmer unfamiliar with the implementation of FSM to know how to correct this unless they are told "what the error really means." It is noteworthy that others have also found that reporting parsing errors is problematic for novices [5].

Figure 3 displays a student's proposed context-free grammar for the language of palindromes (discussed in Section 3). In addition to the context-free grammar, the student includes an example of a word that ought to be successfully derived using an FSM-defined tester. There are four errors in this grammar:

**Capitalization** FSM convention is that nonterminals need to be capitalized and terminals need to be lower case. In this example, the list of nonterminals needs to be capitalized. In the third rule, the B needs to be lower case and the S needs to be capitalized.

**Unknown symbols** The list of rules contains the undefined symbol 'EMP. Presumably, the student meant to use quasiquote, instead of quote, to define this list. The symbol 'ARROW is also unknown, but given that the FSM rule parser ignores the middle component of a rule it is something that in practice has no effect.

```
(define palindrome (make-cfg '(s b)           ;; the set of nonterminals
                             '(a b)           ;; the alphabet
                             '((S ARROW EMP)  ;; the list of rules
                               (S ARROW aSa)
                               (S ARROW Bsb))
                             '(S)))           ;; the starting nonterminal

(grammar-derive palindrome '(a b b a))
```

Figure 3: Student's proposed context-free grammar for $L = \{w \in (a,b)^* | w\ is\ a\ palindrome\}$.

**Starting symbol** The argument for the starting symbol is a list instead of a symbol.

**Incorrect Language** The grammar only generates palindromes of even length.

The initial error reported is:

```
symbol->string: contract violation
expected: symbol?
given: '(S)
```

Once again, we see FSM reporting a parsing error. Although the error reported is concerning the Racket function symbol->string, luckily this time the error indicates a problem with the given argument for the starting symbol and the student changes the list to the symbol 'S.

After making this correction, no errors are reported. The student's example has the following result:

```
"(a b b a) is not in L(G)"
```

Why is the student confused by this? The student is expecting some error to help solve the problem. In this case, FSM ought to provide more help given that the constructor is incorrectly used. For example, one error is that 'S is not in the set of nonterminals, but appears as a nonterminal in the set of rules.

The final example involves removing all unreachable states from a deterministic finite-state automaton. An outline of a student's proposed solution is displayed in Figure 4. The student has successfully implemented a function to compute the unreachable states using breath-first search. The student also provides a sample deterministic finite-state automaton and uses FSM to test if it is equivalent to the result obtained from removing the unreachable states. No errors are reported. There are, however, two type bugs. The constructor make-dfa is not properly used. Unreachable states are neither removed from the list of final states nor are rules involving unreachable states removed. These are dependent type errors that FSM can and ought to handle better. Students should get an error when trying to build machines with rules involving nonexisting states. The same holds true for constructing machines with accepting states that are not in the list of states for the machine.

## 5 Design of FSM Error Messages

The first version of the FSM error-messaging system is designed around building guarded constructors for state machines and grammars. This decision follows from the errors made by students observed by the instructor during the maiden voyage of FSM at SHU. The overwhelming majority of errors revolved around misuses of constructors as those outlined in the previous section. Students expressed a great deal of frustration by having no real way to understand the error messages on their own. This meant

```
;remove-unreachable : dfa --> dfa
;Purpose: To remove the unreachable states from the given dfa
(define (remove-unreachable m1)
  (letrec
      (;reachable-states : (listof Rules)(listof States) (listof States)
       ;                   --> (listof States)
       ;Purpose: To determine the reachable states from the given list of
       ;         states and rules
       (reachable-sts (lambda (lor tovisit reachable)
          ; Accumulator invariants
          ;   tovisit: reachable states whose successors have not been found
          ;   reachable: reachable states whose succesors have been found
          ...))

       (new-states (reachable-sts (sm-getrules m1)
                                  (list (sm-getstart m1))
                                  '())))
    (make-dfa new-states
              (sm-getalphabet m1)
              (sm-getstart m1)
              (sm-getfinals m1)
              (sm-getrules m1))))

(define dfa-test1 (make-dfa '(q0 q1 q2 q3 q4 ds)
                            '(a b)
                            'q0
                            '(q1 q3)
                            '((q0 a q1) (q0 b ds) (q1 a q1)
                              (q1 b q2) (q2 a q1) (q2 b q2)
                              (q3 a q4) (q3 b q4) (q4 a q4)
                              (q4 b q4) (ds a ds) (ds b ds))))

(define remove-unreachable-test1 (remove-unreachable dfa-test1))

(check-expect (sm-testequiv dfa-test1 remove-unreachable-test1) #t)
```

Figure 4: A student's transformer to remove unreachable states from a deterministic finite-state automaton.

students had to see the instructor every week to complete homework assignments and take-home quizzes. Although having office hours packed with students may be the dream of a true educator, having students come mostly to help them debug misuses of FSM constructors was nothing less than disappointing. Students, in fact, eventually learned how to fix errors by memorization rather than by the usefulness of the error messages. Students also realized they all faced, in essence, the same error messages and developed a system to take turns to attend office hours. One group would come to office hours and then this group would communicate the real meaning of an error message to the rest of the class.

It was clear that for the next use of FSM it was necessary to develop meaningful error messages. It is noteworthy that all students were upper level undergraduates. This suggests that meaningful error messages are needed at all levels of the undergraduate curriculum. Some argue that compilers and libraries ought to always generate meaningful error messages [4, 5, 14]. With this in mind, we chose to endow FSM error messages with the following characteristics:

**Vocabulary** The vocabulary of the error messages must match the vocabulary the instructor uses in the classroom. This means that the instructors must make an effort to use the same automata theory jargon when developing machines, grammars, and constructive proofs in class.

**Length** The decision was made to have longer informative error messages. Instead of reporting errors piecemeal (i.e., one error at a time), all errors caused by the misuse of a constructor are reported at once. This is a feature that students on the maiden voyage felt they would find most useful. Furthermore, it makes sense, because the inputs to the constructor are dependent on each other.

**Ordering** For state-machine constructors, the starting state, the list of final states, and the list of rules depend on the list of states, the alphabet, and the stack alphabet (for pushdown automata only). Errors for the non-dependent types are listed first. Similarly, for grammar constructors the errors pertaining to V, the set of nonterminals, and to $\Sigma$, the alphabet, are reported first. The expectation is that listing errors with the non-dependent types first may make determining bugs easier.

**Prescription** The FSM error messages should not prescribe solutions as suggested by Marceau et al. [5].

**Highlighting** A misused constructor must be highlighted by DrRacket when an error is thrown. This means that errors must be thrown by the constructor and not by an auxiliary function used by the constructor.

The implementation of the error-messaging system is completely hidden from the user and is designed to operate on top of the existing FSM library. Constructor wrappers, that test the validity of the input received, serve as guards. If the input received is valid, then the unguarded constructor is called. If the input received is not valid, then the wrapper throws an error and prints a message. In this manner, improvements in the error-messaging system and in the FSM constructors may be developed in tandem.

For the Turing machine constructed in Figure 2, the error messages reported now looks like this:

```
the alphabet sigma must be a list
the given rule a must be a list
the given rule b must be a list
```

These error messages are clearly communicating that the alphabet and the rules are not lists. This clues students into the fact that the wrong types of arguments are being provided as input to the constructor. It is noteworthy that DrRacket highlights the use of make-tm in the file students are running.

For the context-free grammar constructed in Figure 3, the errors reported now look like this:

```
b must be uppercase to be valid for V
s must be uppercase to be valid for V
S must be a symbol in V to be a valid lhs for the rule (S -> Bsb)
The symbols (B) are not part of V or sigma.
S must be a symbol in V to be a valid lhs for the rule (S -> aSa)
The symbols (S) are not part of V or sigma.
S must be a symbol in V to be a valid lhs for the rule (S -> EMP)
The symbols (E M P) are not part of V or sigma.
(S) must be in V to be a valid starting symbol
```

The first two messages indicate that elements in V, the set of nonterminals, are not properly capitalized. The next 6 messages are indicating errors in the list of rules stating that there are elements in the rules that are neither in the set of nonterminal nor in sigma (the alphabet of terminal symbols). For a given rule, left-hand side (lhs) errors are listed first followed by errors from the right-hand side (rhs). The eighth message may be of particular interest. It is explained to students that the right-hand side of a rule is decomposed (i.e., parsed) into FSM symbols. The error is indicating that E, M, and P are not recognized as valid symbols in the grammar. Notice that it is not suggesting that the student may have not used a quasiquote to recognize the FSM constant EMP. Such a suggestion would be improper, because it is equally feasible, for example, that the student simply forgot to include E, M, and P in the list of nonterminal symbols. The final message indicates that the argument provided as the starting nonterminal is not in the in the list of nonterminal symbols. For this final error message, it would be possible to report that he argument must be a symbol. The decision was made to use the displayed format to keep individual messages short. None of the error messages, as the reader may expect, address the fact that the constructed grammar does not generate the correct language.

For the finite-state automaton constructed in Figure 4, the error messages are not empty. The reported errors are as follows:

```
the state q3 in the rule (q3 a q4) must be in your list of states
the state q4 in the rule (q3 a q4) must be in your list of states
the state q3 in the rule (q3 b q4) must be in your list of states
the state q4 in the rule (q3 b q4) must be in your list of states
the state q4 in the rule (q4 a q4) must be in your list of states
the state q4 in the rule (q4 a q4) must be in your list of states
the state q4 in the rule (q4 b q4) must be in your list of states
the state q4 in the rule (q4 b q4) must be in your list of states
(q3) cannot be final states because they are not in the initial
   list of states for the machine
```

DrRacket highlights the error as coming from the use of the constructor make-dfa in the body of remove-unreachable. The first eight messages indicate that states q3 and q4, which are not states in the machine, are being used in the rules of the machine. No corrective action is suggested. Students must decide if these states need to be or not be in the list of states for the machine. In this case, these are the unreachable states and this should help students make progress towards correctly solving the problem allowing them to realize that their transformer must remove rules involving unreachable states. The final message is indicating that nonexisting states are listed as final states. Once again, no corrective action is suggested. The error should help students realize that unreachable states must also be removed from the list of final states.

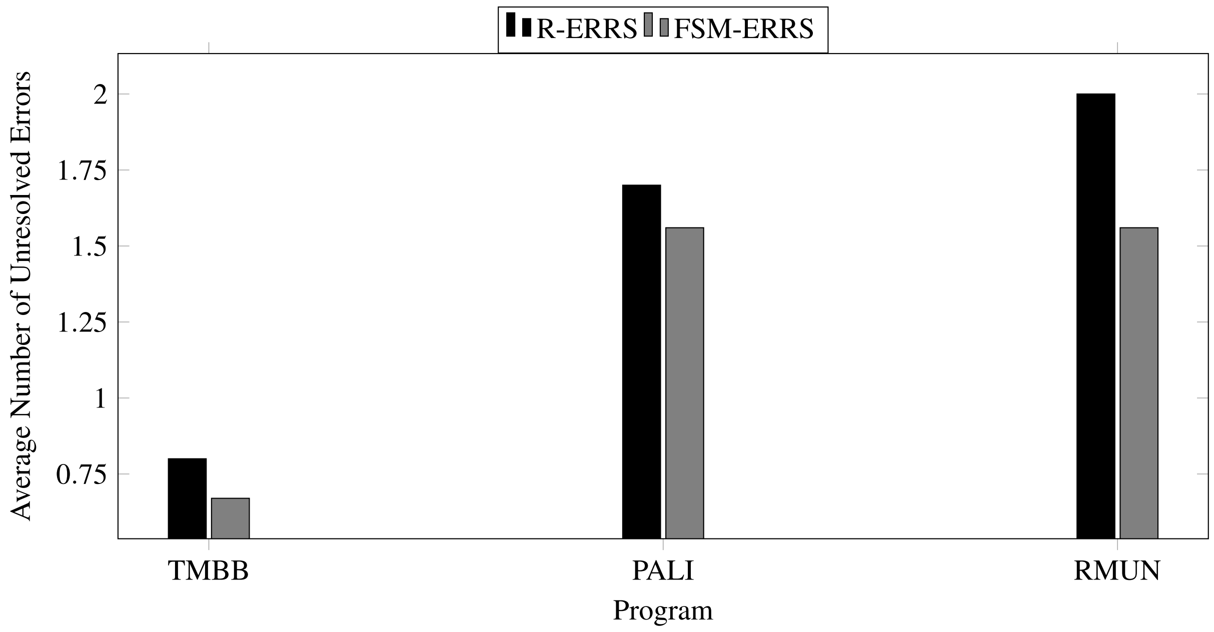

Figure 5: Average Number of Unresolved Errors After Debugging.

## 6 Student Feedback

Students feedback was obtained in two ways. The first method was a control group study to determine the effectiveness of FSM error messages. Students in the second voyage of FSM at SHU in the Spring 2018 semester were evenly divided at the end of the semester into two groups. Each group was asked to debug the three examples described in Section 4. One group was assigned to work with the version of FSM that produces Racket error messages (described in Section 4). The other group was assigned to work with the version that produces FSM error messages (described in Section 5). Of the 25 students enrolled 19 volunteered to participate in the study of which 10 were randomly placed in the group that used FSM with Racket error messages and 9 in the group that used the library that produces FSM error messages.

The second method aims to ascertain student attitudes towards the course, towards programming with FSM, and towards FSM error messages. Students of both the maiden (V1) and the second voyage (V2) of FSM at SHU were surveyed. For the maiden voyage, 100% of the students (8/8) chose to participate in the survey. For the second voyage, 92% of the students (23/25) chose to participate in the survey.

### 6.1 Control Group Study

Figure 5 displays the average number of unresolved errors for each program after debugging. The x-axis represents the programs students attempted to debug. The programs are labeled as follows:

**TMBB** The Turing machine that decides the language of all strings that have bb as a substring.

**PALI** The context-free grammar for the language of palindromes.

**RMUN** The function to remove unreachable states from a deterministic finite-state automaton.

The y-axis is the average number of unresolved errors for each program after debugging. For each program there are two columns. The first is for the control group using FSM with Racket error messages

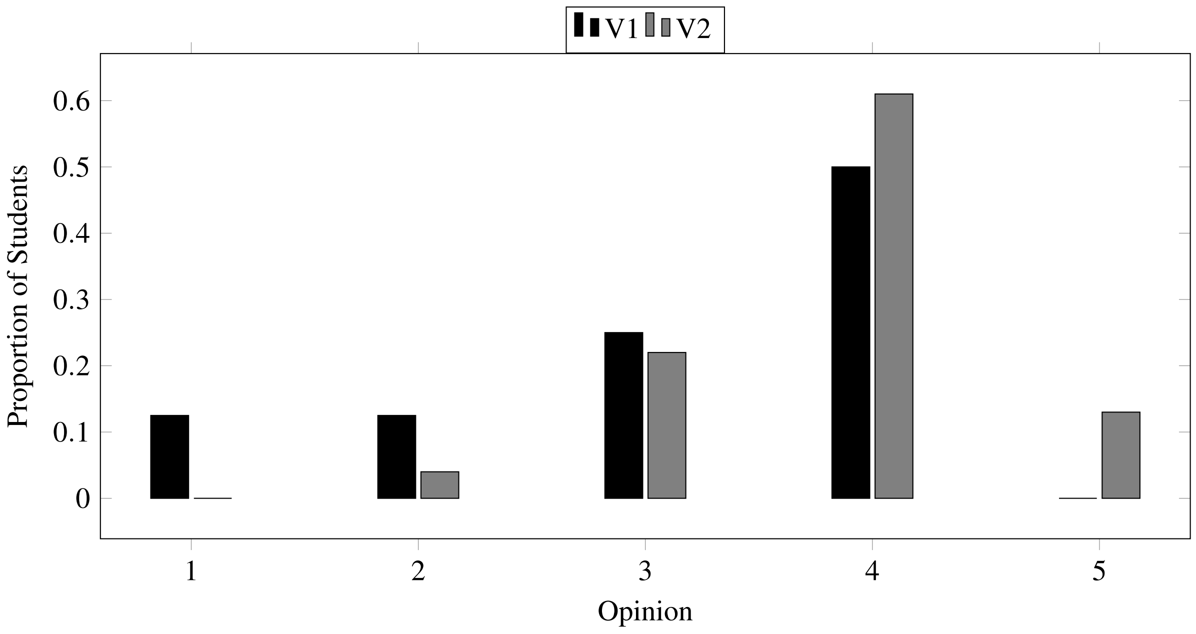

Figure 6: Overall Course Opinion.

(R-ERRS). The second is for the study group using the library with FSM error messages (FSM-ERRS). The students were given one class period (i.e., 75 minutes) to debug the three functions. They were advised to spend at most 25 minutes on each program.

For TMBB, the average number of unresolved errors after debugging is less than 1 for both groups. That is, most students in both groups fully debugged the Turing machine implementation. The majority of students in the study group commented that the FSM error messages were useful in honing in on the problems. In the control group, virtually all students commented that the error messages were "useless." These students also commented that the only reason they were able to debug the function was due to their prior experience using make-tm. In essence, they remembered or looked up in the documentation the correct order for arguments. The study group, nonetheless, exhibits 20% fewer errors left unresolved when compared to the control group. This indicates that FSM error messages have a significant impact on helping students debug programs.

For PALI, both groups had, on average, resolved 2-3 of the 4 bugs. The bug that was most frequently left outstanding in both groups was fixing the language that the grammar generated (i.e., to also generate palindromes of odd length). This is probably not a big surprise as there are no FSM or Racket error messages generated for this bug. The next most frequently unresolved error was the correct capitalization of nonterminals. In both groups, the majority correctly capitalized the starting nonterminal to S in the list of nonterminals and in the rules. In the control group, most students did not change B to b in the rules nor did they remove b from the list of nonterminals. Something interesting happened in the study group. After correctly capitalizing S, the FSM error messages they receive are:

```
b must be uppercase to be valid for V
The symbols (B) are not part of V or sigma.
The symbols (E M P) are not part of V or sigma.
```

The third message was usually the first students tackled. The second message refers to bugs in the rules. The corrective action many students took was to capitalize b in the list of nonterminals. After this corrective action, they did not receive any error messages (if they had already resolved the EMP error) or they received:

```
The symbols (E M P) are not part of V or sigma.
```

Regardless of the order in which reported errors were resolved, most students considered the grammar successfully debugged. Only a handful of students observed that something is still wrong, because they unsuccessfully tried to derive palindromes using FSM's grammar-derive. These students complained that no error messages were reported for palindromes of odd length. This indicates that students are focusing on getting fewer error messages and not on trying to understand if there is a design bug in their grammar. It suggests that instructors would be well-advised to constantly remind students that it does not suffice to only eliminate type errors. Despite these issues, the study group exhibits 9% fewer errors left unresolved when compared to the control group. Once again, this indicates that FSM error messages have a positive impact on helping students debug programs.

For RMUN, students were told that they may assume that the list of unreachable states is correctly computed by the function reachable-states. For this program, we observe the largest gap between the two groups with the study group having on average 29% fewer outstanding bugs. All students in the control group and the majority in the study group, however, did not resolve either of the two bugs. In the control group, only 2 students realized that the rules needed to be filtered and none realized that the set of final states also needed to be filtered to eliminate unreachable states. Eight students stated that they received no error messages and that the machine worked correctly. This indicates that students, do indeed, need informative error messages to find implementation shortcomings. In the study group all but 1 student knew what needed to be done to eliminate the bugs. The majority, however, did not get around to implementing the corrective actions. This indicates that the FSM error messages are effective in finding implementation shortcomings. These students simply needed more time to implement the solution.

## 6.2 Student Survey

Students were asked what is their overall evaluation of Automata Theory course on a scale from *1 = A complete waste of time* to *5 = Beyond Excellent*. The results are displayed in Figure 6. Half, 50%, of the students on the maiden voyage of FSM at SHU highly ranked the course (responses in 4-5). In contrast, the overwhelming majority of students, 74%, on the second voyage highly ranked the course. The only significant differences in the delivery of both versions of the course is the use of the tailored-made error-messaging system and the associated coverage of error messages in class using the error-message vocabulary. In both groups, qualitative responses stated that the best part of the course was the programming component. The students on the first voyage, however, did complain about how often they had to go to office hours. All this indicates that FSM error messages have a positive impact on how receptive students are to the course.

Students were asked to overall evaluate their level of interest in programming with FSM on a scale from *1 = Not at all interested* to *5 = Very interested*. The results are displayed in Figure 7. In the maiden voyage at SHU, only a minority of students, 25%, signaled a significant interest in programming with FSM (responses in 4-5). The overwhelming majority of students, 75%, had, at best, a lukewarm interest in programming with FSM (responses 2-3). In contrast, for the second delivery of the course using FSM error messages an overwhelming majority of students, 65%, signaled a significant interest in programming with FSM and a minority, 31%, signaled a lukewarm interest. This drastic change in observed attitudes can only be attributed to having a better error-messaging system. It is important to

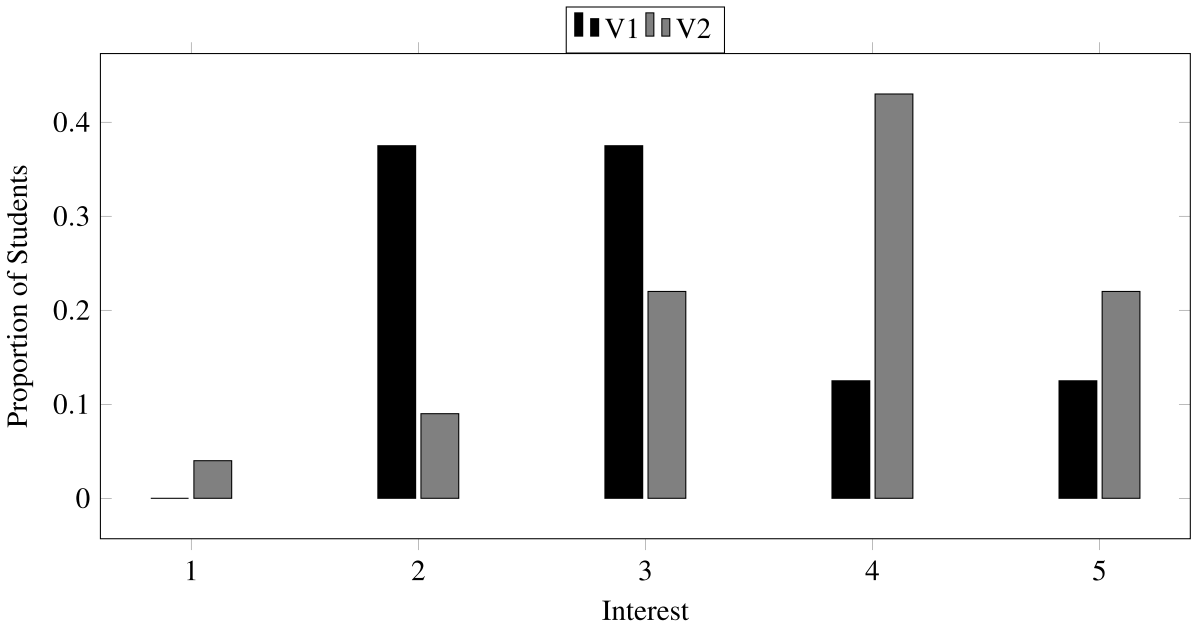

Figure 7: Level of Interest in Programming with FSM .

note that students are specifically expressing an interest in programming with FSM. The lesson that needs to be derived is that even libraries for advanced courses benefit from a well-designed error-messaging system.

Students in the second voyage of FSM at SHU were asked to evaluate their overall opinion of FSM error messages on a scale from *1 = A complete waste of time* to *5 = Beyond excellent*[1]. The results are displayed in Figure 8. The responses clearly suggest that students like the FSM error messages. A majority of students, 57%, signaled a very positive opinion (responses in 4-5). In fact, 96% of students signaled a positive opinion (responses 3-5). This result, in our opinion, is the most important reason that explains why students have a more positive attitude towards Automata Theory and programming in FSM. We also have no doubt that having the instructor cover error messages and use error-message vocabulary in class helped make the error-messaging system a success.

## 7 Concluding Remarks and Future Work

This article presents the design of FSM error messages. These messages aim to help students diagnose the reason for bugs without prescribing a solution. The error messages revolve around type violations when using an FSM constructor. The system reports all errors that are found instead of just the first one. It also specifically avoids reporting parsing errors and list errors for non-dependent types before errors for dependent types. The empirical data collected strongly suggests that the FSM error-messaging system is very successful. It has successfully contributed to increasing students' interest in Automata Theory and in programming in FSM. Furthermore, almost all students report a positive opinion about FSM error messages. The work strongly suggests that an error-messaging system tailored for students is as important in advanced courses as it is in courses for novices.

[1]Students in the maiden voyage of FSM were not asked this question, because error messages were not on our radar.

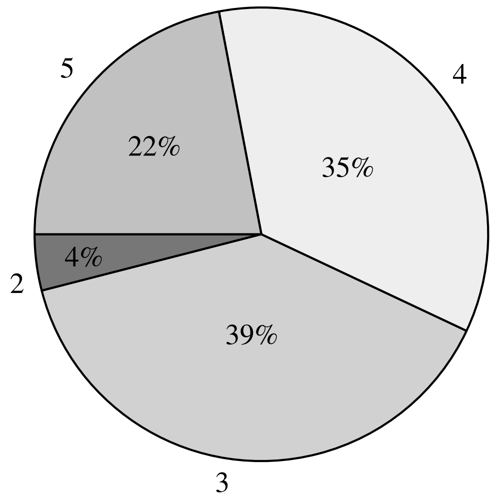

Figure 8: Overall Opinion on FSM Error Messages.

Future work includes exploring how to improve the wording of individual messages. For example, should lots of detail be provided or should less detail be provided? Currently, FSM error messages provide lots of detail when machine rules contain type errors, but much less detail is provided for grammar rules. More qualitative research is needed to determine if one strategy is better than the other. To date, we can see no difference. That, however, may be due to the instructor covering error messages in his lectures. Future work also includes collecting more quantitative data to strengthen or modify the conclusions suggested in this article.

## 8 Acknowledgements

The authors thank all the Automata Theory students that accepted to participate in this study. We also express our gratitude to the Department of Computer Science and to the Office of the Dean of The College of Arts and Sciences of Seton Hall University that made this work and the travel to present results possible.

## References

[1] Marcus Crestani & Michael Sperber (2010): *Experience Report: Growing Programming Languages for Beginning Students*. In: *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, ACM, New York, NY, USA, pp. 229–234, doi:10.1145/1863543.1863576.

[2] James I. Hsia, Elspeth Simpson, Daniel Smith & Robert Cartwright (2005): *Taming Java for the Classroom*. In: *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '05, ACM, New York, NY, USA, pp. 327–331, doi:10.1145/1047344.1047459.

[3] Harry R. Lewis & Christos H. Papadimitriou (1997): *Elements of the Theory of Computation*, 2nd edition. Prentice Hall PTR, Upper Saddle River, NJ, USA, doi:10.1145/300307.1040360.

[4] Guillaume Marceau, Kathi Fisler & Shriram Krishnamurthi (2011): *Measuring the Effectiveness of Error Messages Designed for Novice Programmers*. In: *Proceedings of the 42Nd ACM Technical Sym-*

*posium on Computer Science Education*, SIGCSE '11, ACM, New York, NY, USA, pp. 499–504, doi:10.1145/1953163.1953308.

[5] Guillaume Marceau, Kathi Fisler & Shriram Krishnamurthi (2011): *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*, Onward! 2011, ACM, New York, NY, USA, pp. 3–18, doi:10.1145/2048237.2048241.

[6] John C. Martin (2003): *Introduction to Languages and the Theory of Computation*, 3 edition. McGraw-Hill, Inc., New York, NY, USA.

[7] Linda McIver & Damian Conway (1996): *Seven Deadly Sins of Introductory Programming Language Design*. In: *Proceedings of the 1996 International Conference on Software Engineering: Education and Practice (SE:EP '96)*, SEEP '96, IEEE Computer Society, Washington, DC, USA, pp. 309–316, doi:10.1109/SEEP.1996.534015.

[8] Marco T. Morazá & Rosario Antunez (2014): *Functional Automata - Formal Languages for Computer Science Students*. In James Caldwell, Philip K. F. Hölzenspies & Peter Achten, editors: *Proceedings 3$^{rd}$ International Workshop on Trends in Functional Programming in Education*, *EPTCS* 170, pp. 19–32, doi:10.4204/EPTCS.170.2.

[9] Jonathan P. Munson & Elizabeth A. Schilling (2016): *Analyzing Novice Programmers' Response to Compiler Error Messages*. *J. Comput. Sci. Coll.* 31(3), pp. 53–61. Available at `http://dl.acm.org/citation.cfm?id=2835377.2835386`.

[10] Marie-Hélène Nienaltowski, Michela Pedroni & Bertrand Meyer (2008): *Compiler Error Messages: What Can Help Novices?* In: *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '08, ACM, New York, NY, USA, pp. 168–172, doi:10.1145/1352135.1352192.

[11] Phil Race (2005): *Using Feedback to Help Students Learn*. https://www.heacademy.ac.uk/knowledge-hub/using-feedback-help-students-learn.

[12] Paul A. Schiliep (2015): *Usability of Error Messages for Introductory Students*. *Scholarly Horizons: University of Minnesota, Morris Undergraduate Journal* 2(2), p. Article 5. Available at `http://digitalcommons.morris.umn.edu/horizons/vol2/iss2/5`.

[13] Michael Sipser (2013): *Introduction to the Theory of Computation*, 3rd edition. Cengage Learning.

[14] V. Javier Traver (2010): *On Compiler Error Messages: What They Say and What They Mean*. *Adv. in Hum.-Comp. Int.* 2010, pp. 3:1–3:26, doi:10.1155/2010/602570.